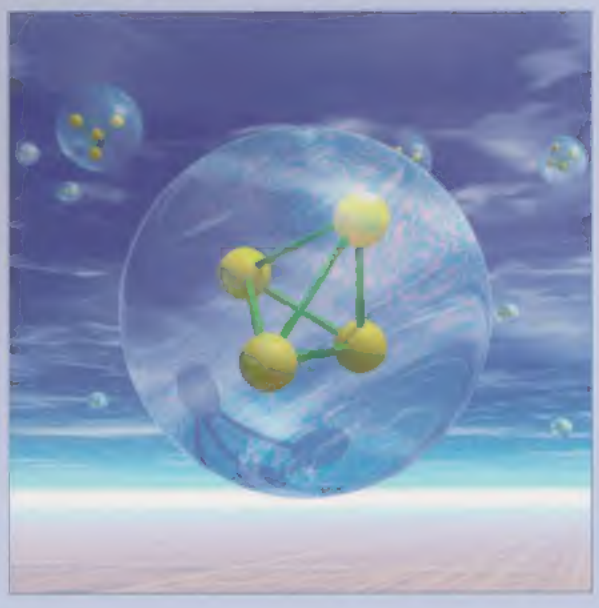
The Open University

M255 Unit 1
UNDERGRADUATE COMPUTING

Object-oriented programming with Java



Object-oriented programming with Java

Unit 1

M255 Unit 1
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

Object-oriented programming with Java

Unit 1

This publication forms part of an Open University course M255 *Object-oriented programming with Java*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom; tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at http://www.open.ac.uk where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit http://www.ouw.co.uk, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom for a brochure: tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall
Milton Keynes
MK7 6AA

First published 2006. Second edition 2008.

Edited and designed by The Open University.

Typeset by The Open University.

Printed and bound in the United Kingdom by The Charlesworth Group, Wakefield.

ISBN 978 0 7492 5493 3

2.1

The paper used in this publication contains pulp sourced from forests independently certified to the Forest Stewardship Council (FSC) principles and criteria. Chain of custody certification allows the pulp from these forests to be tracked to the end use (see www.fsc.org).

CONTENTS

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland, Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Critical Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Author and Critical Reader

Robin Walker, Critical Reader

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Compositor

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

Thanks are due to the Desktop Publishing Unit, Faculty of Mathematics and Computing.

Introduction

Welcome to the first unit of M255 *Object-oriented programming with Java*!

As the course title suggests, the emphasis of the course is on object-oriented programming – writing software from an object-oriented perspective. Object-oriented programming is concerned with constructing computer systems out of interacting units of software, called objects. Objects know nothing of how each other work, but they can interact (when a program is executing) by sending messages to each other. As you'll see later, one of the most powerful aspects of object-oriented programming is that the code that produces interacting objects can be reused and interchanged between programs, so increasing programming productivity.

Programming in an object-oriented language is more than just learning new syntax rules; it requires a new way of thinking. The idea is not to concentrate primarily on the fundamentals of procedural languages – data structures and algorithms – but instead to think in terms of the objects that will carry out the required tasks.

The programming language you will use in M255 is Java. However, the purpose of the course is not to teach you the minutiae of the Java language, but rather to teach you fundamental object-oriented programming concepts and skills that will be transferable to any object-oriented language. Hence, while you will certainly learn quite a lot of Java, and write lots of program code, we will be concentrating on those aspects of the Java language that best demonstrate object-oriented principles and good practice.

The best way to learn any language is to practise using it. Learning a new way of programming is no different, so you will find that this course has many practical programming activities for you to carry out! In working your way through the course and engaging in all the activities you will gain a good understanding of object-oriented principles, and a solid grounding in the use of the Java programming language.

After a brief review of fundamental hardware and software concepts (Section 2), this unit introduces the basic elements of object-oriented software (Section 3) and presents a short history of the Java programming language (Section 4). In Sections 5 and 6 you will begin to explore objects by engaging in computer-based activities.

Since M255 is a Level 2 course, the course team has assumed that you already have some programming experience, such as that gained from previous study or work, and are familiar with common programming constructs such as loops, if statements, assignment statements and variables.

See the *Course Guide* for a fuller description of prerequisite knowledge.

1 Components of M255

The most obvious component of M255 is the series of printed units (you are reading *Unit 1* at the moment!). However, as described below, there is more to M255 than these printed units.

- All units include *computer-based activities*; practical sessions involving the use of your computer. The details of what you need to do for each activity, and a discussion of the results, are contained at the appropriate points in the printed units.
- *Email* is used for sending messages to and receiving them from your tutor and other students. Your computer will need to be linked online to a *network* (probably via a modem) and have the appropriate communications software running for email to work. (Your computer will also need to be online and running appropriate software to use the next two components described below: conferences and web pages.)
- *Conferencing* is supported by the FirstClass system. Your tutor-group conference, together with your regional M255 conference, will be a focus for general academic discussion during your study of M255. You should use them to discuss questions and issues about the course with your fellow students. Depending on circumstances, sub-conferences devoted to particular topics may be created within these conferences. Remember, however, that your tutor is the person you should contact with specific academic queries: you should not use your tutor group conference as a means to contact your tutor on some specific issue – you should either email your tutor directly, or make contact via phone or letter.

  Sharing and discussing ideas about the course with fellow students can be an exciting and rewarding experience and you are encouraged to make full use of your tutor-group conference. Please note that there are some basic rules about behaviour when using FirstClass. These are described in the Conditions of Use sub-conference, which is available within the OU Service News conference that is on your desktop.

  It is important that you access FirstClass at least once a week to look at the postings in your tutor-group conference and check for message in your MailBox.
- *Web pages* on the M255 website give you access to other components of the course, such as a study calendar, additional learning materials, assignments, news information (for example, to correct errors or clarify points in material), electronic versions of some unit printed texts, further explanations on a topic, and references and hyperlinks to further reading. It is important that you access the M255 website at least once a week to check for announcements on the news page.
- *Course software* is distributed via CD-ROMs and includes the FirstClass client and BlueJ, the software you will use to program in Java.

Before continuing we suggest you install BlueJ and the other course software if you haven't already done so (refer to the *Software Guide* for full details).

2 Fundamental hardware and software concepts

Before embarking upon the main focus of the course – object-oriented programming – we will take a look at some fundamental hardware and software concepts. In this section you will see what is meant by terms such as hardware, software, systems, applications and programs and then go on to look in more detail at how computers are capable of functioning so flexibly.

2.1 Hardware and software

Hardware consists of the tangible parts of the computer system – the parts that can be kicked. Examples of hardware include the electronic circuits inside the casing of your computer such as the central processing unit (CPU) and main memory, and also peripheral devices. A **peripheral device** is any component of the computer that is not part of the essential computer (i.e. the CPU and main memory). The most common peripherals are input and output devices such as the keyboard and monitor, and storage devices such as hard disks and CD/DVD drives. Some peripherals, such as hard disks, are usually mounted in the same case as the processor, while others, such as printers, are physically outside the computer and communicate with it via a wired or wireless connection.

Software, on the other hand, is more abstract – it is a general term for all the applications, programs and systems that run on your computer, that is, it covers everything you cannot kick! Software consists of sets of instructions that tell a computer (or rather the hardware) how to perform a particular task. Examples of software are word-processor applications such as Microsoft Word, browsers such as Microsoft Internet Explorer or Netscape, and communications software such as FirstClass.

Although software and hardware are very different in nature, they are also inextricably related. Any instruction performed by software can also be built directly into hardware, and instructions executed by hardware can often be simulated in software. So there is a trade off. One could build a computer without any software; it would do just one task – but very quickly. However, we expect computers to do a multitude of tasks: calculate our tax returns, write a letter, play chess and maybe surf the Web. Hence it is usual to get the computer hardware to do a lot of very simple tasks (such as adding or subtracting two binary digits), and write software to combine these simple tasks into various sophisticated applications.

2.2 Software: systems, applications and programs

Although software is held as magnetic or optical patterns on a physical object (such as a CD-ROM, DVD, memory stick or the hard disk), software itself is intangible. You cannot see or touch software. Software is written using a programming language, and pieces of text in such a language are often called **source code** or just **code**. This code is then compiled into a sequence of zeros and ones, that is, **binary digits** or **bits**, which make up the instructions and data that the hardware can execute. It is not generally useful to consider software in terms of binary digits being interpreted by hardware as instructions to the computer and few programmers need to think at the bit level. When programmers

Compilation is explained in Subsection 2.4

do discuss software in these terms they are taking a **low-level** view. By this we mean that they are considering the minute detail of how a hardware device performs a task.

To build a better understanding of what is meant by the word 'software' we need to consider how we can categorise different types of software and look at the terms 'system', 'application' and 'program'.

Systems

The term 'system' has subtly different meanings depending on how it is used, as can be seen in the list below.

(a) An **operating system**, as in 'How do I configure my system to allow me to use my new scanner?'.

(b) A **computer system** (a combination of hardware and software), as in 'My system crashed four times last night. I can't figure out whether it is a hardware problem or that shareware game I picked up from a magazine cover disk'.

(c) A **software system** (usually a large piece of software) is essentially meant to run forever (it has no start point or end point) and has to respond to a variety of events that may occur in an unpredictable order. The system is likely to be composed of a number of smaller units of software, called applications, which communicate with each other. For example, 'The patient monitoring system has eight subsystems, not including the part that checks that the others are functioning within normal operating parameters'.

(d) **System software** is categorised as software that helps the computer carry out its basic operating tasks. It is software which is required to support the production or execution of applications but which is not specific to any particular application. System software typically includes:

- the operating system that controls the execution of other programs;
- user interface software such as graphical windows and menus systems or text-based command line interpreters;
- development tools, such as compilers, for building other programs;
- utility programs (involved for example in sending data to a printer or communicating with peripheral devices).

For most of the time we use the term **system** to capture the idea of a large piece of software, as in (c) above. Such a system may be made up of many parts and may be accessed by users in different ways and for different purposes. Occasionally, when talking about hardware or operating systems, we use the term in the sense of (b) – a **computer system**. That is, the combination of hardware and software (predominantly the operating system) providing the technological context for the software programs in which we are interested.

You have probably been a user of a large software system, for example, an airline seat-reservation system. An airline seat-reservation system allows online enquiries and the booking of airline seats from a vast worldwide network of travel agents (and perhaps booking from your own home if you have the appropriate connections to the system). The system at the heart of the reservation system is intended to run for 24 hours a day, forever, and to provide real-time access to the database that identifies the available seats on relevant flights. As a user of such a system you may not always be aware of the other uses that the airline companies (and the travel trade) make of the complete system (of which seat reservations are but a part). Clearly, such a system must also know about the availability and capacity of the aeroplanes being used and their movements around the world.

The meaning of the term 'operating system' as in (a) above more or less matches our idea of a (software) system. Unless you switch off your computer or the operating system crashes, this (operating) system should run forever.

Programs

The notion of a **program** assumes a pattern of 'input data' 'process data' 'output data'. That is, the software that is the program has a starting point at which it takes some input, it then performs whatever computation is needed, and it has an end point at which output is given and the software ceases to run. This contrasts with systems which run forever. However, a system might well call upon the services of a program (via the operating system) to accomplish some simple task. For example, one very simple program might display the numbers 1 to 10 on your computer screen in quick succession. Another might calculate the conversion of pounds sterling into US dollars. Programs may often, but not always, be 'home brewed', that is written by the computer user to solve a specific small task. In this course you will be writing programs.

Exercise 1

Say whether you think the following are programs or systems according to the meanings given to these terms in this unit. Give a reason for each of your answers.

(a) Software that converts a temperature expressed in Fahrenheit to Celsius.

(b) Software to control machinery for cutting timber into sheets of wood veneer.

(c) Software that checks that a timber-cutting machine is correctly calibrated.

(d) Software that issues tickets for the trains running through the Channel Tunnel.

(e) Software that calculates the mean score on a particular tutor-marked assignment.

Solution

(a) This is a program: it takes a number (a temperature in Fahrenheit) as the input, and outputs a number as a temperature in Celsius. Thus it conforms to the input–process–output pattern.

(b) Software needed to control machinery is most likely a system: in essence it is meant to run forever and respond to events related to the machinery or to the materials being processed. (In practice, systems may sometimes have to be stopped and restarted for the maintenance of the machinery.)

(c) Software that checks the settings of a machine is probably a program: it would take readings as input and output suggested adjustments to the machine's settings. The software may be part of a larger system.

(d) Ticketing software is usually a system or part of a larger system: unless it ran forever train operators would lose money.

(e) Software that calculates the mean is a program: it inputs scores, computes the mean and outputs the result.

In the solution to Exercise 1 we have speculated about whether software might be part of a larger system: for example, the calibration software in (c) could well be part of the system specified in (b). It is usual for the distinction between a program and a system to be blurred like this, especially as the system is made up from parts that are themselves programs or systems.

Applications

You can liken **applications** to virtual computers each with a special operating system. For example, when you use a word-processor application you turn part of your general-purpose computer into a computer that knows only about documents and the commands that are relevant to them. When you use a web browser application you turn your personal computer into one that spans the world. A web browser knows about the Web and how to display the documents that reside there. As a computer user you can start up these different computing applications, may have more than one running at the same time, and can switch between applications as required. Think about the situation where you start up a word processor and then, without exiting from it, you start up a web browser. Each application is a virtual computer with its own set of commands, or the same commands with slightly different meanings (for example, the commands Open and Save have different meanings in a word processor and a web browser).

Applications differ from [illegible] designed to run forever and they generally [illegible] single task for a single user. For example [illegible] computer can only be used by the user [illegible] the production of text documents.

Note that the terms program and application are often used synonymously.

Exercise 2

System software is categorised as software that helps the computer carry out its basic operating tasks, a software system is software that is meant to run forever and has to respond to a variety of events that may occur in an unpredictable order, and application software is categorised as software that helps the user carry out a task by means of the computer. Using these categories, describe software that:

(a) allows the user to print material on a printer;

(b) maintains a personal calendar and address book;

(c) monitors and controls the temperature inside a school.

Solution

(a) Since printing is a basic operating task, this software is categorised as system software.

(b) This software is considered an application: it turns your computer into a specialised computer – a personal assistant – that, when active, can remind you of birthdays and meetings.

(c) This is a software system: it is designed to run forever. It may well make use of several programs to open and shut valves and to monitor temperature periodically.

In practice, it can often be difficult to categorise software as either an application or a software system: how complex does an application need to be before it can be called a system? The boundary between the two can be very blurred. In a similar manner it can often be difficult to categorise software as either a program or an application. Hence you will find that these terms tend not to be used too precisely!

2.3 The operating system

A computer's operating system defines the computing experience. It is the first software that you are aware of when you turn on the computer, and the last software you notice when the computer is shut down (unless it crashes!). Yet most computer users cannot

say with any certainty precisely what it is that the operating system does, so it is worth spending some time getting this clear

An operating system (OS) is the software responsible for the control and management of hardware and basic system operations (such as data input and output), as well as running application software such as word-processing programs and web browsers Common operating systems for personal computers include Linux, Mac OS (for the Apple Macintosh) and the various versions of Windows, e g Windows 2000 and Windows XP

In essence an operating system acts as an intermediary between the user (or an application program) and the computer hardware, as shown in Figure 1. It essentially enables the user to carry out a variety of complex tasks on the computer, without the need to know anything about what goes on 'inside the box'.

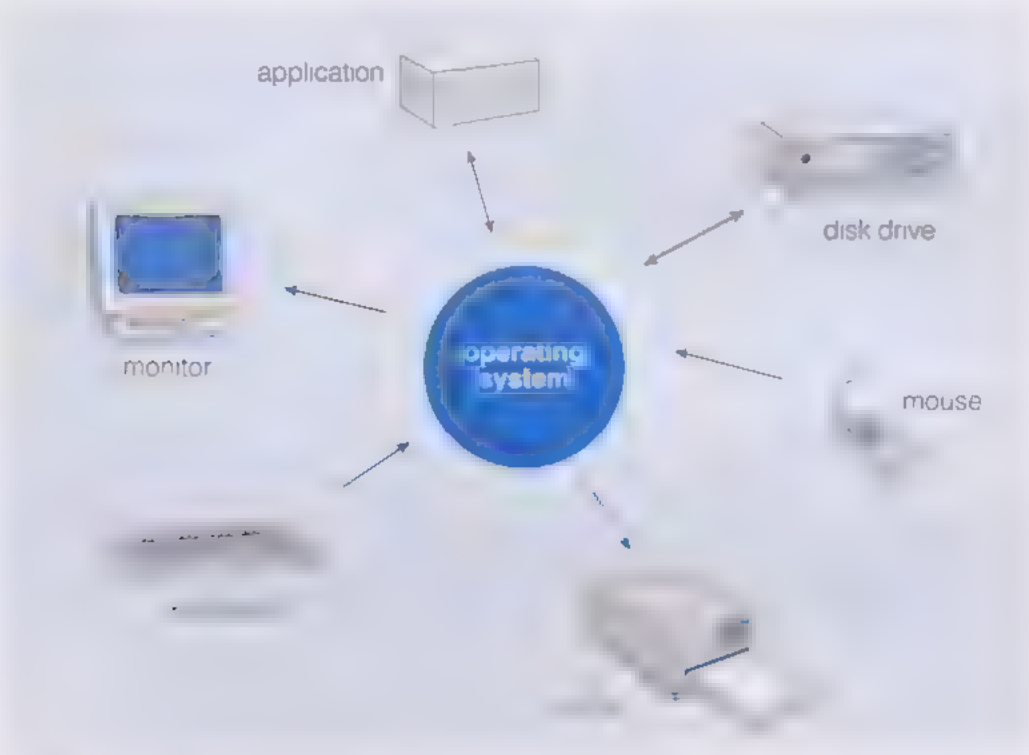


Figure 1 The operating system

Of course, not all computer systems require an operating system. A computer that controls the fuel-injection system in a car does not need an operating system It has one task to perform and unchanging hardware to control Since the computer simply runs a single program, there is no need for the program to be stored magnetically or to be loaded into RAM An operating system is necessary because it allows the same computer hardware to be used for many different applications and purposes, with the details of how the hardware works controlled by the operating system

The next seven short subsections will explain more about the work of the operating system and will explain how it is loaded when your computer is first switched on

Management of memory

During the execution of a program, data and instructions are stored in the computer's main memory It is the job of the operating system to allocate an appropriately sized area of memory to each program (or application), and to ensure that program instructions and data do not interfere with each other, or with the data and instructions of other programs

Coordination and control of peripheral devices

In order to carry out its tasks a computer may need to communicate with one or more peripheral devices. For example, it may wish to receive input data from the keyboard or mouse, read from a file on a storage device, send output to the monitor or printer, and connect to a network. The operating system coordinates all these operations, ensuring that data is moved safely and efficiently between the different components of the system.

Scheduling of access to the processor

The operating system [illegible] by programs to be run and ensuring that the processor is used efficiently. For example, if the currently running program finishes, or is interrupted in order to wait for data from the hard disk, the operating system will ensure, if possible, that another program is given access to the processor.

Provision of basic utilities

Operating systems also provide basic utilities such as disk formatting facilities, file management systems and software installation wizards.

Provision of an interface between applications/programs and hardware

An important part of the operating system is that it provides a consistent way for software to communicate with the computer's hardware without having to access that hardware directly or know about the details of the hardware, or even the minutiae of the processor's specifications. The operating system provides this through an application programming interface (API), which is a set of high-level instructions (a protocol) [illegible] such as printing a file or saving a file to disk. So, for example, instead of an application program itself containing [illegible] the operating system through the API to print the file. The operating system then communicates with the printer to carry out the request. This abstraction allows a software developer to write an application on one computer and have a high level of confidence that it will run on another computer with the same operating system even if the amount of memory, makes and types of peripherals are different on the two machines.

Provision of a user interface

The user interface is the part of the operating system that communicates with you, the computer user. It provides a means of inputting data and instructions, and presents output in an understandable way.

Widget is the term used to describe components such as windows, buttons and sliders that are used in GUIs

The user interfaces of early operating systems, such as CP/M and DOS, were text based (termed command line interfaces), requiring the user to learn a set of commands, which needed to be typed in following precise rules. Output to the screen also consisted entirely of text. Today, all personal computer operating systems provide graphical user interfaces (GUIs), although most also provide (often hidden away from the novice user) a text-based interface.

GUI-based operating systems (of which the various versions of Microsoft Windows are the most common examples) make use of icons, menus and other widgets with which the user interacts via a pointing device, usually a mouse. Most people find graphical interfaces more intuitive, quicker to learn, and easier to use than sequences of textual commands. A further advantage of GUIs is their availability for use by programs other than the core software provided by the operating system. For example, programmers of

[illegible] such as word processors or spreadsheet packages, do not have [illegible] programs from scratch. They can use the operating system's API to 'hook in' to the GUI. In addition to providing a consistent, indirect way for application programs to communicate with the computer's hardware, the API also provides high-[illegible] the creation of windows, buttons and menus, so making quite much easier for the programmer. This ensures that all applications making use of the GUI components have a consistent 'look and feel', which in turn makes it easier for users to learn how to use new applications.

Booting your computer

When you switch on a computer, the first thing it needs to do is to load an operating system (which is usually stored on the hard disk).

[illegible] there's a *boot program* which is implemented directly in hardware [illegible] read-only memory (ROM). This program is stored [illegible] ROM [illegible] during manufacture and is permanent. It cannot be overwritten and will not disappear when power is lost to the computer. The boot program is executed [illegible] the computer is first switched on and it will typically run a test of the [illegible] other peripherals are connected to the system before [illegible] The boot program is a short program to load a larger program is called bootstrapping which comes from the idea of someone pulling [illegible] computer has given rise to the expressions 'booting up' and 'rebooting' a computer.

Software which is stored in a ROM is called *firmware*. It cannot be changed easily.

SAQ 1

For each of the following functions of an operating system give one reason why the function is important.

(a) Managing the allocation of memory

(b) Providing a user interface.

(c) Scheduling access to the processor

ANSWER

(a) Memory allocation ensures that [illegible] each other or with the data and instructions of other programs.

(b) User interfaces enable the user to communicate with the computer.

(c) [illegible] use it efficiently.

2.4 How programs execute on a computer

When writing software it is necessary to express a solution to a problem in a programming language resembling a limited natural language that can be understood and interpreted by human beings. That solution (as written in a programming language) is called the source code. It must then be translated into a primitive language called machine code – in effect the bits that can be understood and executed by the hardware of the computer. Since the programming language abstracts away from the details of the machine code language we call the former a **high-level language** and the latter a **low-level language**. There are a number of models for this process. One such model can be depicted as follows.

We often use the term 'code' without the qualifier 'source' or 'machine' and rely on the context to give the correct meaning. Except for this discussion, we shall not be interested in machine code in this course, so mostly we use 'code' to mean 'source code'.

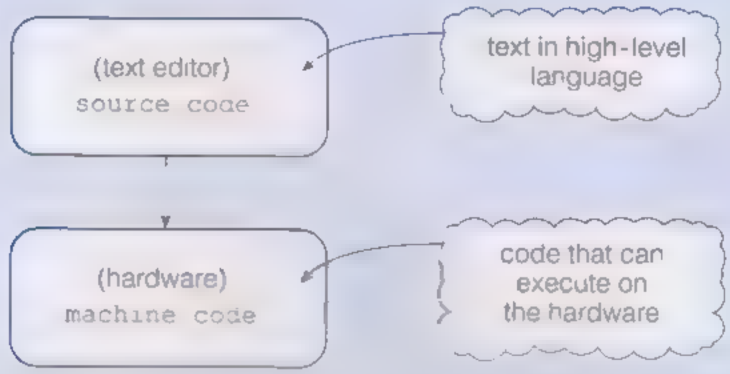


Figure 2 Relationship between source code and machine code

Translating [illegible] language source code to a low level machine code program is usually carried out by a piece of software called a **compiler**. Translation (but not into a program) can also be done by an **interpreter**, which will translate source code line by line into machine code as and when it is required.

During compilation a compiler must first check that the text conforms to the syntax rules of the language, that is, it is properly formed. Only if this check does not show up problems does the compiler proceed to produce the machine code that will be executed. Typically, compilers [illegible] generate some additional information not given in the [illegible] the **run-time system**. It enables the [illegible] is made, that is at **run-time**. [illegible] computer system (or platform), that is, to a particular combination of hardware and software.

The major problem with this simple model of compilation is that the compiled code is not portable to other machine architectures, as different machine types employ different machine code languages. If you wish to move your software onto another architecture, for example from a PC to a Macintosh, you would have to recompile the high-level language source code with a Macintosh specific compiler to produce Macintosh machine code (see Figure 3 below).

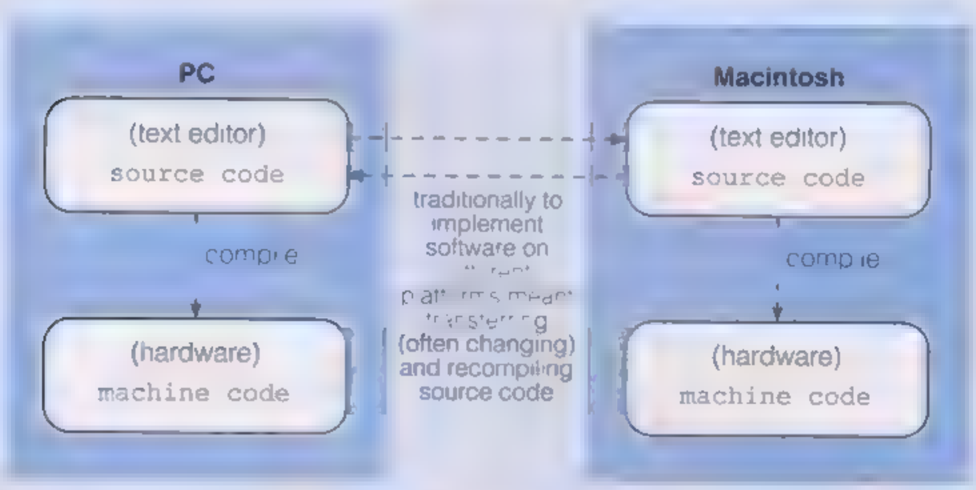


Figure 3 Compiling code for different computer systems

Another more portable mode of compilation makes use of a special layer of software (or even real machine architecture) called a **virtual machine** (VM). In this model of compilation the high-level language source code is not compiled to some architecture dependent machine code. Instead it is compiled to an **intermediate code**, that is to the machine code of the notional (virtual) machine. In Java environments the intermediate virtual machine code is usually called **bytecode** as it is organised into 8 bit bytes. Using this intermediate code approach allows low level essentially executable code to be moved unchanged between different computer systems (see Figure 4).

There are other language environments besides Java which also compile to bytecode

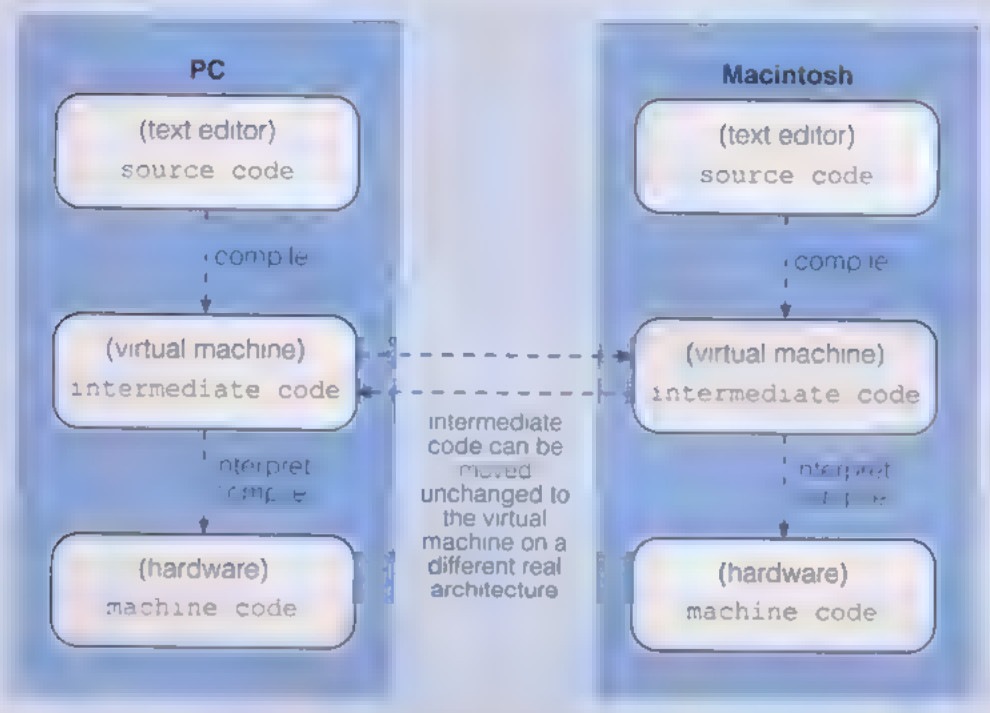


Figure 4 Compiling for a virtual machine

One [illegible] the software (now in intermediate code) on a real computer.

The [illegible] **interpreter** [illegible] software [illegible] virtual machine [illegible]. Every time an application is run the interpreter takes the [illegible] (intermediate code) and translates each notional instruction, one at a time, into the real instructions for the real computer hardware to execute. This is a relatively slow process because so much software is involved in interpreting, whereas the machine code from a simple compilation can be executed directly by the hardware. The advantage of using an interpreter is that your program can be executed on different real computer systems if that is required.

The second option is for the virtual machine to include another phase of code generation in which all the intermediate code for a piece of software is translated into the real machine code in one go, so that it is ready for the real machine hardware to execute. This is the traditional approach where intermediate code is used. It originates from a time when programs were prepared and tested as a whole rather than in component parts, and has the disadvantage that the programmer must wait for all of the intermediate code to be translated into machine code before the program can be executed and tested.

In programming environments where the second option is used, the programmer may never be aware that a second translation has taken place

This sort of approach is also called **just-in-time compilation**, for the obvious reason.

The third option is a combination of the first two and is called **dynamic compilation**. This option is particularly attractive when software is developed in relatively small chunks – modules that can be separately compiled. This is the option used by the **Java Virtual Machine** (JVM). When a request is made to compile a chunk of code, the environment's (in this course's case Bluej's) built-in compiler produces intermediate code (bytecode) for that chunk of code. This is then compiled into machine code by the virtual machine software when the code is first executed, and this real machine code is stored for subsequent executions. Therefore subsequent execution of that code has all the speed that results from simple compilation. (Of course, any code that is never executed will never be translated into machine code.)

In summary, the advantage of [illegible] that makes use of a virtual machine is that it ensures that [illegible] that machine it was compiled can be translated [illegible] computers so long as each computer has the correct virtual machine.

2.5 The computer as a layered device

In this section we have defined and described the terms hardware and software (including various sub-categories), the operating system and the Java Virtual Machine. To bring all these things together it is useful to consider a computer system as a layered device. So, for example, a Java program runs on top of the Java Virtual Machine, which runs on top of the operating system, which itself runs on top of the hardware.

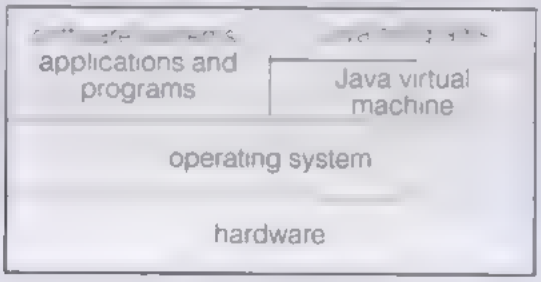


Figure 5 The computer as a layered device

Without the layers of software in modern computers, computer systems would not be as useful and popular as they are today. While the complexity of these underlying layers has increased greatly in recent years, the net effect has been to make computers easier for people to use.

3 Object technology

Commercial programs are large, very, very large. They typically consist of hundreds of thousands of lines of code, sometimes millions. As with all complex systems whether or not these systems involve computers, they need to be organised in such a way that the human mind can comprehend and deal with them. Comprehension is helped if it is possible to view a complex system as made up of simpler parts that interact in an overall structure. Throughout the history of software development there has been an active search for useful structuring techniques and for programming languages that support such techniques.

This chapter describes one structured approach to programming that uses collections of interacting objects to build a more complex whole. It is called object-oriented programming. After looking at the shortcomings of procedural programming, we provide a brief introduction to object-oriented programming and introduce some of the terminology used in the rest of the book. We set the scene for the section by seeing how object oriented techniques have developed over the last 40 years.

3.1 Procedural programming

To understand **object-oriented technology** it is useful for us to look at what went before. Until fairly recently the predominant method for structuring programs was **procedural programming**. This involves breaking down the program into a collection of procedures, each of which carries out a particular task. A procedural program has a structure of a main program, involving a number of procedure or function calls, which can in turn call further functions or procedures.

An algorithm is a detailed sequence of actions to perform to accomplish some task (named after the ninth century Arab mathematician, Al-Khwarizmi)

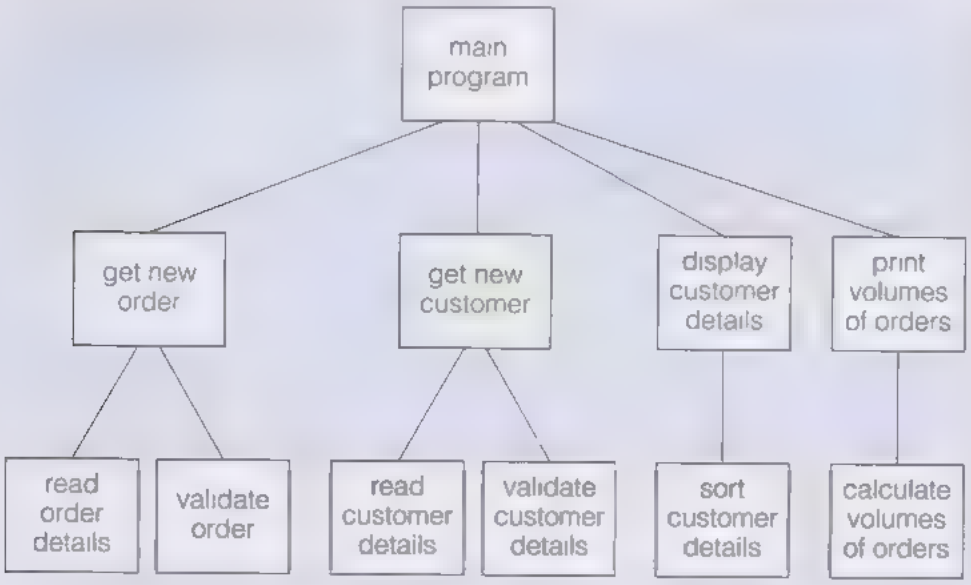


Figure 6 The structure of a procedural program

In such a design, data is of secondary importance and is placed into separate structures (called data structures). Often this data is global to the whole program so is visible and accessible to every function or procedure in the program (see Figure 7 below), each of which will be able to change that data.

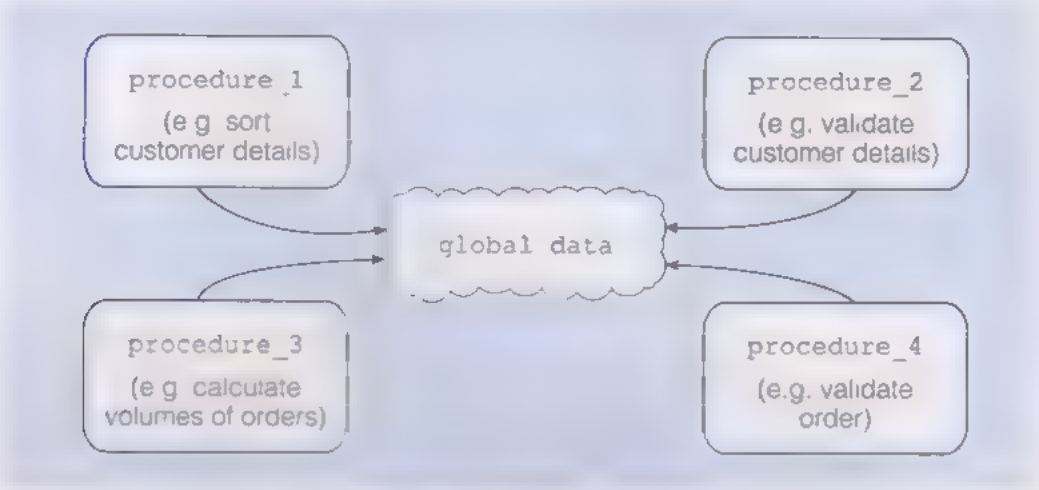


Figure 7 Procedures accessing global data

The ramification of so much data being global is that if a change is made to the format of any data structure, all the functions and procedures that operate on that data will also have to be modified to reflect the new data type. Thus one small change has a knock on effect throughout the program, involving changes to numerous widely scattered routines.

3.2 Object-oriented programming

The idea of viewing software and the need for designing and writing software in terms of objects is not a new one. The idea has been around for 40 years, but its value has only really become evident in the last fifteen years or so. Not all software has been designed around the concepts of object-oriented programming, but most software with user interfaces of windows, buttons, menus and similar icons has been built using objects.

In the object-oriented approach to software design all the processing carried out by software is considered as being done by objects. You are already familiar and comfortable about the concept of an object in the physical world, being surrounded by them everyday – cars, people, toasters, DVD players, managers, etc. In object oriented programming we use *software* objects to **model** real-life ones. The software objects simulate the part of the real world (often called the **application domain** or **problem domain**) with which we are concerned. For example, if we need a system to help manage patients in a hospital (the **domain**), our software will in some sense have to construct representations of part of the world of hospital administration. The objects we might need are those that deal with modelling real-world patients, doctors, nurses, wards and so on. Since the system would be a computerised one we would also need objects for the user interface, such as windows, menus and buttons.

In the real world, people, organisations and even machines interact with one another by exchanging requests and passing one another information. In the same way, object-oriented programs consist of code for creating objects that can communicate by sending **messages** to each other and they may get responses back, called **message answers**, from other objects.

There are two important aspects to an **object**: its **attributes** and its **behaviour**. An attribute is some property or characteristic of an object: so a patient object might have attributes such as 'condition', 'date admitted', 'medication' and so on. The **attribute value** of 'condition' might be 'malaria' and the attribute value of 'date admitted' might be '3/1/2006'.

The behaviour of an object is the collection of actions an object knows how to carry out. Each object has a list of messages it knows how to respond to. An object modeling a patient might need to know how to respond to messages such as 'take medicine'. The message 'take medicine' might well be sent by a nurse object.

For human users of a software system we cannot interact directly with software objects. We communicate with them via a *user interface*. Actions such as clicking the mouse or pressing a key on the keyboard will cause messages to be sent to the appropriate objects.

Objects work to achieve some end: objects need to collaborate with each other. Sometimes they need to delegate work to others – a bit like teamwork. Using the hospital example, a 'doctor' object may send a message to a nurse object requesting it to 'take medicine' to the 'patient' objects. The nurse object would then in turn send 'take medicine' messages to those patient objects.

Figure 8 is a picture of what's happening inside an object-oriented program when it's running. The program, in Figure 8, is a collection of objects sending messages to each other. Each object is like a cell. To get an object to do something, you send it a message. The arrows between the objects show messages being sent. An object needs to know what behaviour will result from receiving a message. The internal structure of the object receiving the message provides the behaviour. This illustration is limited by the need to show arrows as being fixed: in software that's running, the messages are sent (and received) as required in response to other messages.

Figure 8 Object-oriented software is a collection of objects sending messages

In its basic definition, an object is an entity that contains both data (in the form of attribute values) and behaviour (the actions it takes on receiving messages). The word *both* is the key difference between object-oriented programming and the more traditional procedural approach. In a well-programmed object nothing outside the

object can directly change the value of the object's attributes. Indeed the only way to get an object do anything (including perhaps, changing the values of some of its attributes) is to send it a message. This is one benefit of the object-oriented approach because an object is responsible for updating its own data (the attributes) any changes to the structure of that data only affect that type of object. Since it is usual for software to be changed and developed over time, this is an important benefit and there are further benefits, such as reuse, that will emerge during the course.

Attributes and state

[illegible] A bank account object might [illegible] it may keep a track of its [illegible] For example balance and [illegible] the two of the attributes of a bank account object: the attribute balance might have the value 100.00 and the attribute creditLimit might have the value 400.00.

For most programming languages it is usual to have to run together multi-word names or *identifiers* as they are known, such as creditLimit into single words. This will shall prepare you for programming by running together the words [illegible] to form the word [illegible] using a single upper-case letter to mark the start of the second original word. More generally, identifiers use a single upper-case letter to mark the start of each word after the first, for example: `startOfRace` and `firstPastThePost`.

The values of an object's attributes together determine the object's **state**. For example, the state of a bank account object at a [illegible] comprises the values of its attributes – the value of `balance` may be `100.00` and value of `creditLimit` may be `400.00`. Figure 9 is a diagrammatic representation of a bank account object. The [illegible] contents of the two sections within the diagram. The object has a 'memory' (the values of [illegible] message names are omitted).

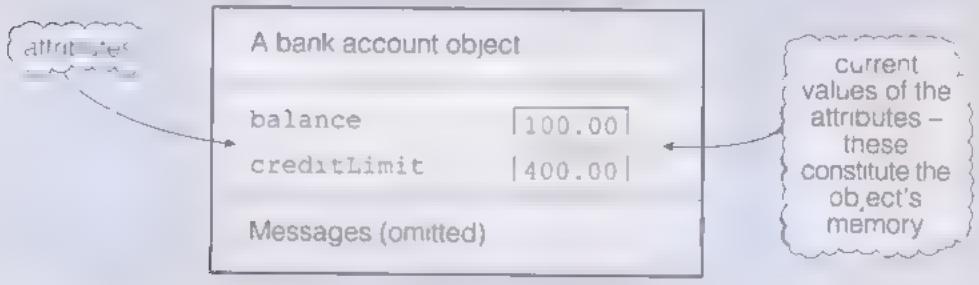


Figure 9 Diagram of a bank account object, showing its attributes and attribute values

SAQ 2

What is the difference between the attributes of an object and the state of an object?

ANSWER

Attributes describe the kinds of information that an object needs in order to provide the required behaviours. The state of an object is the particular data held by all the attributes at a given time, that is the attribute values. For example, a bicycle object may have the attributes manufacturer and size. Its state is described by the values of these attributes, perhaps Raleigh and 21.

Messages

As mentioned earlier, the only way of getting an object to do something is to send it a message. To change some part of the state of a bank account object (that is to change the value of one of its attributes) – for example to increase its balance – a message must be sent to the object. Similarly, to find out the value of an attribute, for example to find out the balance of a bank account object, a message must be sent to the object. On receipt of a message, a check is made that the object has been programmed to understand that message. If it has, the object will respond with the requested action, such as changing the value of one of its attributes or returning some other value. Sometimes the requested action may involve the object sending messages to other objects. You can picture the object with the messages it understands, as well as the information it holds (as the values of its attributes), as illustrated in Figure 10 below.

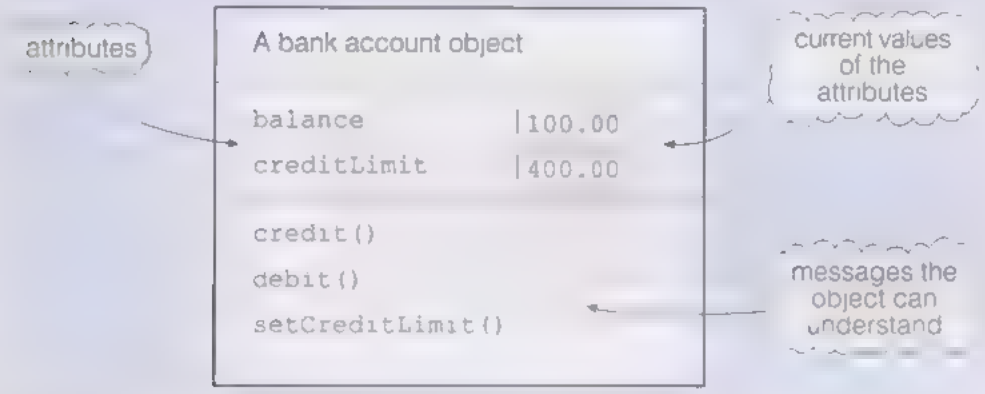


Figure 10 Diagram of a bank account object, with a partial list of the messages it understands

In the above bank account object the message `credit()` would increment the value held by the attribute `balance`, the message `debit()` would decrement the value held by the attribute `balance`, and the message `setCreditLimit()` would set the value held by the attribute `creditLimit`. If the attribute `creditLimit` had a value of `400`, the message `debit()` should fail if it tried to take the `balance` below `-400`.

Note the use of a pair of round brackets (parentheses) after the message names – all message names in Java are followed by these brackets. Sometimes additional information (called an **argument**) needs to be put inside these brackets when a message is sent. For example to debit 50 pounds from the bank account object you send it the message `debit(50)`. You will learn more about arguments in *Unit 2*.

Just as you saw with the names of attributes, in Java the name of a message must be a single word and therefore `set credit limit` would not be allowed. So, to preserve the meaning of the phrase, while obeying the rules of Java, we run the words together and use a capital letter to show where a new word would have begun.

3.3 A short history of object-oriented technology

The idea of object-oriented software originated in Norway in the mid-1960s with the language Simula, an extension to the Algol programming language.

Algol (ALGOrithmic Language) was a procedural language, designed in the late 1950s, for programming scientific calculations.

Simula was designed to make it easier to write programs that simulated real-world phenomena such as industrial processes. It allowed complex systems such as a North Sea oil terminal to be simulated (and so managed) in software. Programmers could manipulate objects that combined information and behaviour in single units of software. For example, adding a valve between two pipes in a Simula model of an oil refinery simply involved creating a new valve object, setting its operating parameters, and linking it to the appropriate pipe objects. The new valve object brought with it the ability

to be opened and closed, altering the flow of oil appropriately. If the same refinery were modelled using a conventional procedural programming language, the various behaviours associated with the valve (such as opening and closing) would probably be distributed around the program and in various procedures, and therefore harder to find and change.

The next major development of these ideas (building systems from components that keep together information and behaviour) [illegible] at the Xerox company's Palo Alto Research Center (PARC) in Northern California in the 1970s and early 1980s and is largely identified with Alan Kay and Adele Goldberg. They developed the first truly object-oriented programming language [illegible] initially aimed at children, on the assumption that if they could use it [illegible]. Indeed, many of the ideas that define modern computing were also developed at PARC: the ideas of a personal computer, graphical user interfaces (windows and menus), laser printers, local area networks, the list is impressive.

Many [illegible] companies that became the [illegible] computers, but the object ideas as embodied by Smalltalk were not so readily adopted. Although seminal to object te[illegible] many large complex systems, Smalltalk failed to [illegible] widespread success in the software industry. The reasons for this were threefold:

- ▶ Smalltalk development environments were relatively expensive to buy.
- ▶ Compiled Smalltalk programs run on a virtual machine rather than directly on hardware and so were relatively slow running on the computers of the 1970s and 1980s.
- ▶ [illegible] languages of the day, so few in the industry were prepared to make such a big change.

Unix is a time-sharing operating system implemented almost entirely in C. By 1991, Unix had become the most widely used multi-user general-purpose operating system in the world.

However, [illegible] which was used extensively [illegible] operating system. Indeed C is still frequently used today, and in the mid 1980s two languages appeared that embodied objects and were based on the C language.

- ▶ In 1985 Bell Labs released C++ (written by Bjarne Stroustrup) that added objects to the C language.
- ▶ In 1986, the StepStone Corporation released Objective-C (written by Brad Cox), which was a combination of C and Smalltalk syntax. Objective-C gained an early success being adopted in 1988 as the development language for the short-lived NeXT computer and its (Unix-based) NeXTstep operating system. Currently Objective-C is used as the principal programming language for Apple's Mac OS X.

Additionally, Eiffel, a purely object-oriented language, also made its debut in 1986. It introduced a number of features, including the ability to generate documentation automatically from source code. This feature found its way into Java.

By the 1990s C++ had emerged as the market leader in object-oriented languages. Its popularity was due, in part, to compatibility with the large existing base of C programmers and the widespread use of Unix, which runs on many different types of machines.

Although C++ created many converts to object-oriented ideas, it does have a major drawback. It is what is termed a hybrid language, a procedural language that has had the capabilities for object-oriented programming bolted on. The ramifications of this are that it is possible for a programmer to write in an object-oriented style, or a procedural style, or a mixture of both! This can (and does) result in complicated, hard-to-follow code that is difficult to maintain. It also had the result of programmers fooling themselves that

they were writing object-oriented code just because they were using C++ when in fact they were writing the same old procedural code.

In the next section you will learn about the development of the Java programming language, a language which was built from the ground up. While its syntax superficially resembles C, it is a fully-fledged object-oriented language owing more to the spirit of Smalltalk than to C.

4 The origins of Java

In many ways it's a sheer accident that we are writing and that you are studying a course about object oriented programming that uses Java rather than some other language. Object-oriented languages have been with us for decades and other languages notably Smalltalk C++ Eiffel and Objective C were until the late 1990s the dominant languages used in object oriented software development In this section we shall look at the history of the Java language and try to answer the question as to why Java is today such a popular language The answer to this is one of convergent technologies, and serendipity

4.1 In Switzerland

In the late 1980s scientists at the European Particle Physics Laboratory (usually known as CERN short for the French version of the name **C**onseil **E**uropeen pour la **R**echerche **N**ucleaire) were having problems accessing and sharing documents electronically Documents were stored on a variety of servers in a variety of incompatible formats This made the retrieval and viewing of documents problematic – which server was a particular document on? What software was needed to read it? If a document referred to another document what server was that document located on? Problems were compounded at CERN because of the nature of the site visiting academics and students needed to get up to speed on experiments very quickly Furthermore there were many collaborations of physicists who were remotely based around the world If these scientists wanted to share documents with colleagues based in CERN they had to organise and format them so that they would be compatible with the main CERN computing systems Not surprisingly many researchers were unwilling to expend the extra effort to make their work conform to the CERN system

Tim Berners-Lee a software engineer at CERN proposed a solution based upon the theoretical work of Vannevar Bush who back in the 1940s had described a theoretical system for storing information based on associations and the work of Ted Nelson and Douglas Engelbart who respectively first coined the phrase hypertext and developed a successful implementation of hypertext in the 1960s Hypertext allows documents to be published in a nonlinear format enabling the reader to jump instantly from one electronic document to another

In a markup language the text (and images) are surrounded by special text (or *tags*). These tags, may be interpreted by software to generate a display

In brief Berners-Lee proposed a distributed hypertext system or web that would run over the Internet Documents would be formatted with a simple markup language and then uploaded to computers running his proposed server software Any person with a computer connected to the Internet would be then able to read (using his viewer software) those documents from anywhere in the world What's more a document on one server could have links to any number of other documents whether on the same server or dispersed on servers around the world to which the user could jump to with a single click of a mouse.

Berners-Lee began work to develop this information system in 1989 By 1990 he had written the Hypertext Transfer Protocol (HTTP) the language computers would use to send hypertext documents over the Internet, and designed a scheme to give documents addresses on the Internet calling these addresses Universal Resource Identifiers (URIs) By the end of the year he had also written a browser application to retrieve and view these hypertext documents He called this first ever web browser

WorldWideWeb. Hypertext pages were formatted using the Hypertext Markup Language (HTML) that Berners-Lee had written. He also wrote the first web server. A web server is the software that stores web pages on a computer and makes them available to be accessed by other computers on the Internet. Berners-Lee set up the first web server, known as 'info.cern.ch', at CERN.

In 1991 he made the source code for his WorldWideWeb browser and the web server available freely on the Internet so that others would be encouraged to set up web servers. The one limitation of this was that all the software was written for NeXT computers running the NeXTstep operating system. However by making the code freely available he hoped that others would make the software (both browser and server) available on other operating systems. So began the story of the World Wide Web.

4.2 In the USA

In 1990 Patrick Naughton, a disgruntled Sun Microsystems software engineer who was about to leave Sun Microsystems for its rival NeXT, detailed in a letter to Sun Microsystems management the shortcomings of the company's software division, along with his own growing appraisal of NeXT's software, named NeXTstep operating system. Shaken by Naughton's perceptive assessment of the problems their software division faced, Sun Microsystems's management commissioned [illegible] by James Gosling and three others to form a research group to investigate something new and exciting which would allow them to catch the 'next wave' of computing.

In early 1991 the group met and decided to look at the application of computers to consumer electronics devices. At this early stage the team were considering such household items as VCRs, fridges, microwave ovens and washing machines, and thinking about the possibilities of developing, say, a central control unit (possibly handheld) for all these items. In this way the group evolved the concept of a network of different [illegible] (home security, entertainment units, etc.) that could all pass messages between themselves. As a part of the project was to design [illegible] This task fell to James Gosling, [illegible] to the conclusion that the [illegible] addressed by creating an entirely new language.

Given the application, the language needed to have the following characteristics:

- *Familiarity* – the C and C++ languages were widely used in consumer electronics so basing the syntax of the new language on these existing ones would aid acceptance and hence use.
- *Platform independence* – the concept was to have a range of devices (from different manufacturers) communicating with each other and thus the language would need to be able to perform on a variety of processors. This characteristic meant the language would have to be an interpreted language that could be run on virtual machines located on each device. Under this scheme bytecode containing the appropriate instructions could be produced on one device (for example the central control unit), then sent around the home network for execution on the virtual machine residing on the device requiring control (see Subsection 2.4 if you need to remind yourself how an interpreted language is used).
- *Robustness* – for consumer acceptance the new technology would need to run without failure. Thus the underlying language technology should omit various error-prone features of C and C++, and incorporate strong in-built syntax checks.

- *Security* – as the various devices would be exchanging information within their network, the language would need to prevent intrusion by unauthorised code getting behind the scenes and introducing viruses or invading the file system.
- *Object orientation* – as the architecture of object-oriented languages fits so well with the architecture of client/server systems running over a network, the language should be designed to be object-oriented from the ground up. In a client/server system, software is split between server tasks and client tasks. A client sends requests to a server asking for information or action, and the server responds. This is similar to the way that objects send messages to each other and get responses (message answers) back.

The design and architecture [illegible] were drawn from a variety of languages including Eiffel, Smalltalk, Objective [illegible] Mesa, and Gosling completed an interpreter for the language by August [illegible] the language Oak, apparently after the tree that grew outside the window of his office (although other stories abound).

None [illegible] the use of bytecode and virtual [illegible] SD Pascal languages. However, what was unique was that all the above features came together cleanly in a single language.

While Gosling had been working on the language, other members of the team had been working on the hardware side and, in August 1992, the team demonstrated a prototype remote control like device with a touch sensitive screen called *7. When a user first touched the screen, it displayed a cartoon world where a character named Duke (shown in Figure 11) guided the user through a cartoon representation of the rooms of a house.

Figure 11 Duke – the cartoon character that first appeared in the *7 (© Sun Microsystems)

Everything was done without a keyboard – a user navigated through the house by gliding their finger across the remote's screen to interact with the various devices. For example, by sliding a finger across the screen, the user could pick up a virtual TV guide on the sofa, select a movie, drag the movie to the cartoon image of a VCR, and program the VCR to record the show. The senior management at Sun Microsystems were ecstatic: this was revolutionary for 1992 and, in that November, a subsidiary of Sun Microsystems called FirstPerson Inc. was set up to further develop and market this new technology.

Despite great expectations, commercial success did not follow; there was no real market for such devices (the technology was too far ahead of its time) and the world was not ready. The future looked a little uncertain for the emerging language, so a race was on to find a new application for Oak. In early 1993, the team heard of a Time-Warner request for proposals for a set-top box operating system, including on-demand interactive technology. FirstPerson Inc. worked at developing a TV set-top box based on Oak to coordinate the transmission of video, data and money securely over a distributed network. They presented the prototype to Time-Warner but unfortunately lost the contract to their rivals in Silicon Valley – Silicon Graphics. That was the last straw – after one too many failures, Sun Microsystems dissolved FirstPerson Inc. and assigned the employees to various other projects within the parent company. It looked very much like Oak was destined to be consigned to the dustbin of history.

4.3 The technologies come together

In the meantime, since 1991, Tim Berners Lee's brainchild, the World Wide Web (WWW) had gone from strength to strength because Berners Lee had made the source code public.

As the number of users on the Web grew it became more attractive as a medium. Scientists, who were already used to sharing information on the Internet, began to embrace the Web. It was easier to post information on the Web once than to reply repeatedly to multiple requests for the same data. They also no longer had to worry whether or not the other scientists [illegible] operating system as new browsers were developed. Government [illegible] to make their information public, also began turning toward the Web.

Berners Lee had developed [illegible] NeXT personal computer. As the potential of the Web [illegible] began creating new browsers for Mac, PC and Unix users. For instance, students at the Helsinki University of Technology wrote Erwise for Unix machines. Pei Wei, a UC Berkeley student wrote Viola and colleagues of Berners Lee at CERN wrote a browser for Mac machines called Samba.

The Web grew exponentially, both in the number of sites and users. The number of visitors to the original web server (info.cern.ch) grew by a factor of ten every year. By the summer of 1993, the original site was getting ten thousand hits a day. However, at this point the Web was still the preserve of mainly scientists and academics. The reason for this being that the first browsers were rather complicated to use, and the documents (web pages) that these early browsers could read were either all text or just a single image [illegible] images, video and text could not be displayed on the same web page (very different from the situation today), and thus the Web was not of much interest to the general public.

This all changed [illegible] University of Illinois NCSA (National Center for S[illegible] Andreessen together with a team [illegible] The browser [illegible] NCSA Mosaic [illegible] as it offered the user a [illegible] continued their programming [illegible] history of the World Wide Web was reached when they released free versions of the Mosaic browser for the Macintosh and Windows operating systems. Not only could the browser display text, graphics and video clips on the same page (and play audio), but crucially the browser was relatively easy to use and available for three popular operating systems. This browser could display text, graphics and video clips on the same page and play audio. The World Wide Web had become multimedia and the online community liked it; in fact liked it very much. No longer was the Web an environment for dry scientific documents, it now became a virtual world full of colour and moving images. Subsequently Andreessen and most of his team left NCSA to form the Mosaic Communications Corp., which later became Netscape and made them all multimillionaires!

Until that point, the Web had been totally overlooked by large corporations such as Sun Microsystems and Microsoft. However, public reaction to Mosaic convinced a few key members of the original Sun Microsystems team that Oak could play a part in the Web explosion. After all, Oak was platform independent, ran on a virtual machine and was designed to run over a network – albeit a network of toasters and fridges and televisions. However, if a browser were written that incorporated an Oak virtual machine, any Oak program residing on a web server could be executed on a browser that incorporated an interpreter. Such applications could make the Web experience far more interactive and

more important y lead to commerc al exp oitat on of the Web An Oak program running w th n a web browser wou d be ab e to query a database take customer deta s and take online payments – lessons learnt from the TV set-top box prototype A eureka moment had been reached, and the race was back on

n September 1994 Patr ck Naughton wrote a prototype browser ca ed WebRunner that incorporated an Oak virtual machine. The idea that a browser could support Oak app cat or s (ca ed apc e's excited many and WebRunner was the perfect p atform from which to demonstrate the power of the ar guage Unfortunate y a patent search revea ed that Oak was a ready a trademark and so the story goes the team came up with the replacement name – Java – during a trip to a coffee shop Thus Oak was renamed Java and WebRunner renamed HotJava

The first pub c release of Java and HotJava browser came on 23 May 1995 at the SunWorld conference. The announcement was made by Gage the Director of Science for Sun Microsystems His announcement was accompanied by a surprise announcement by Marc Andreessen, Executive Vice President of Netscape, that Netscape would be including Java support in its browsers As Netscape was, at the time, the dominant supplier of Internet software, this gave the Java language a major boost and significant cred bility ... so much so that Microsoft soon followed suit and implemented Java support in Internet Explorer From then on Java's popularity as a programming anguage grew meteoric ly and t has now grown nto a full-scale deve opment system capable of being used for developing large applications that exist outside the Web environment

5 Speculating about objects

In this section we shall ask you to carry out a number of activities using the application StarOffice that is supplied on the Online Applications CD-ROM. The purpose of you carrying out these activities is so that you can visualise the object-oriented ideas discussed so far in this unit. StarOffice can be used for both word-processing and drawing graphics. These tasks are so familiar that we usually take them for granted and do not stop to think that they work by using object-oriented technology but as we shall see, objects are very much involved!

The idea that a piece of software is constructed from hundreds or thousands of objects telling each other what to do by sending each other messages is important. In this section you will see how sending messages causes objects to behave in particular ways.

5.1 Objects in a StarOffice document

If you have not already done so, please install StarOffice, which is supplied on the Online Applications CD-ROM. You can find details of how to do this in the booklet inside the Online Applications CD-ROM case.

This subsection comprises a series of activities using StarOffice. If possible you should complete it in a single session. These activities aim to reinforce the concepts introduced in Section 3 that an object has a state (its memory, which is made up by the values of its attributes) and that it reacts (its behaviour, or how it acts or responds to messages) which may depend on its state. What you see on the screen are things that you can apparently see (such as rectangles, words and buttons), but the things you see are visible representations of objects. The activities concentrate on the objects that make these actions possible, such as buttons and windows.

As you carry out the activities, you should watch for these object-oriented ideas. Sometimes they may not be immediately obvious to you, but they will be discussed further in this section. In particular you should think about the memory that an object needs – its attributes, the values of which make up an object's state.

ACTIVITY 1

Launch StarOffice with a new drawing document by selecting **All Programs|Star Office 8 StarOffice Draw** from the Start menu (or, if you have StarOffice open, select from the File menu, New and then Drawing, as illustrated below in Figure 12).

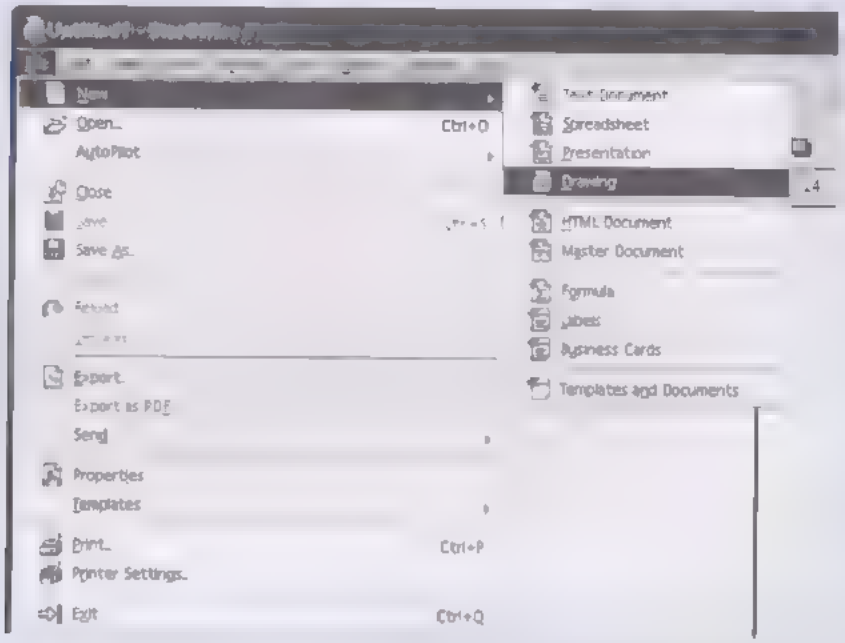


Figure 12 Opening a new drawing document in StarOffice

You will then see an empty drawing document with toolbars arranged along the top and bottom of the drawing pane (the window into which you can draw objects).

Assume that you will be able to draw shapes including lines in the empty drawing pane and spend a short time thinking about the sorts of object that might be involved in a StarOffice drawing document.

You are now going to draw a rectangle and move it. But, before you do this, think about the objects involved. At this stage of the course guesses are perfectly all right – the important thing is to think about what is going on. Remember that as well as the shapes you are drawing there are also objects such as buttons that are used for communicating with the application. The following steps will guide you through what you need to do to draw and move a rectangle.

1 Note which of the buttons on the left-hand side toolbar is selected when the new drawing document is first opened and what shape the cursor is when it is over the drawing pane.

2 Familiarise yourself with some of the buttons on the bottom toolbar, more specifically, counting from the left, buttons two to four. Move the mouse pointer over each button. Leave it for a few seconds and a label (a 'tool tip' message) will appear.

3 Find the button in the bottom toolbar for drawing a rectangle. Click the button and move the cursor to the drawing pane. Note the changes to the button and the cursor. Which objects do you think might have changed state?

4 Click on one position in the drawing pane, hold the mouse button down and drag the cursor to another position – a rectangle is produced when you release the mouse button. The rectangle remains selected (shown by the green blocks, which are sometimes called 'handles'). Which objects do you think might have changed state?

5 Deselect the rectangle by clicking anywhere on the drawing pane outside the rectangle. The blocks will disappear. Which object do you think might have changed its state?

6 Note the colour of the rectangle and the colour and thickness of the lines that make up the rectangle. These are three attributes of the rectangle object.

Making sure that the rectangle is still selected, change its colour from blue to red by selecting Red from the dropdown menu (that at present should be labelled Blue 8) in the toolbar immediately above the drawing pane.

7 You are now going to move the rectangle. Select your rectangle (click once inside it). When the cursor changes to an icon looking like the four points of a compass, 'drag' the rectangle to another position on the screen. Which object(s) might have changed state?

To drag an object, move the mouse with the (left) button in the down position.

DISCUSSION OF ACTIVITY 1

1 In the bottom toolbar, the first button (carrying an arrow icon) is selected as indicated by its white shading and dark blue edging, and the cursor in the drawing pane is an arrow shape.

2 Button two is for drawing lines, button three is for drawing arrows and button four is for drawing rectangles. You can ignore the other buttons for now.

3 The Rectangle button has changed state – it is now shaded. The cursor has changed state – its shape has changed to a crosshair and box to remind you that you are about to draw a rectangle. The state of the drawing pane must also have changed – it must now remember that it should draw a rectangle when you drag the cursor across the drawing pane.

4 The state of the drawing pane object must have changed. It now contains a rectangle, an object that has various attribute values that make up its state.

5 When first drawn, the rectangle was selected, shown by the green handles. Now we have deselected it, and the handles are no longer displayed. The drawing pane remembers what objects are or are not selected.

6 The default colour of the rectangle is blue (specifically Blue 8) and the lines that make up the rectangle are black and the default line thickness is 0.00. Later you will see how to change these attribute values. The state of the rectangle object must include something about its position (in the drawing pane), the colour of its edges (black), the thickness of the edges (0.00), its size and its fill colour (which you changed to red).

7 Noting that the rectangle's position is part of its state, it is the rectangle whose state has changed. A second possibility, which you may have considered, is that the drawing pane remembers the whereabouts of the objects it displays. However, it is clearly more sensible for the rectangle to remember its own properties (including where it is) than for the drawing pane to have to remember all the properties of all the objects that are in it. Indeed, the whole point of having objects is so we can give them the properties (attributes) that belong to them rather than having a single complex object that remembers everything.

In this example, we have taken the view that the information about which objects are currently selected is part of the state of the whole drawing pane, but it is in fact quite possible that we could make each object remember for itself whether it is selected currently or not.

ACTIVITY 2

If your rectangle from Activity 1 has been deleted, before starting this activity draw another one.

1 Work out how to draw an ellipse and do so. Notice that it is selected. Deselect your ellipse. **Which object has changed its state and how is this shown?**

2 Select the rectangle and notice how the cursor changes as it hovers over the handles and when it is inside the rectangle. What happens when you click and hold the (left) mouse button down on one of the handles and drag the handle?

DISCUSSION OF ACTIVITY 2

1 Deselecting the ellipse returns it to its unselected state. The drawing pane has changed its state by adding the ellipse to its list of shapes. This fact is shown by both a rectangle and an ellipse appearing. This is not obvious because drawing a rectangle or an ellipse like this appears to be so 'natural', so like what you might do with pen and paper. However, there is no magic or 'naturalness' involved in software: you interacted with the application that created the appropriate object in the drawing pane and arranged for the shape to be shown in the drawing pane. You can think of the drawing pane as an object that has an attribute whose value is a list of the objects that are 'in' the drawing pane. Every time we draw a new shape it is added to this list. Thus the state of the drawing pane has changed.

2 When the cursor hovers over the handles it changes shape to a two-headed arrow to indicate in which direction(s) the shape will be stretched. To resize a drawing object, you drag one of its handles.

When the cursor is in the middle of an object it changes to the four-points-of-the-compass icon to indicate that dragging will *move* the object.

ACTIVITY 3

A piece of text can be typed into a drawing as a graphical object.

Click on the Text button (the sixth button from the left in the bottom toolbar). With the Text button selected, click and hold down the (left) mouse button in the drawing pane to mark one corner of a rectangular area and, with the mouse button down, drag the cursor to another position marking the opposite corner of the area. Release the mouse button and then *don't touch the mouse*. You now have two cursors: the cursor that looks like the four points of the compass and a text insertion cursor (a thick vertical line positioned top left in the new text box). Without touching the mouse, start typing. When you type, the insertion position is given by the text insertion cursor inside the text box.

What attributes do you think a text box has?

DISCUSSION OF ACTIVITY 3

Working with a text box is slightly different from working with a rectangle or ellipse because you need to be able to type inside the text box a text insertion cursor is shown. Once you have typed a piece of text (a series of characters) you can go back and change it by inserting the cursor anywhere inside the typed text.

A text box has several observable attributes: content (the text displayed in the box), position, width and height.

When you type in a StarOffice document, you are inserting character objects into the document and they appear in the order in which you type them.

In the next activity you are asked to undertake a number of tasks with a StarOffice *text* document.

ACTIVITY 4

Open a new text document by selecting from the File menu New and then Text Document You will then see an empty text document with toolbars arranged along the top and bottom of the text pane (the window into which you can type characters) For the purposes of this activity we are only interested in the toolbar second from top, above the text pane Figure 13 shows the elements of this toolbar that are of interest for this activity.

If StarOffice is not running you will need to launch it

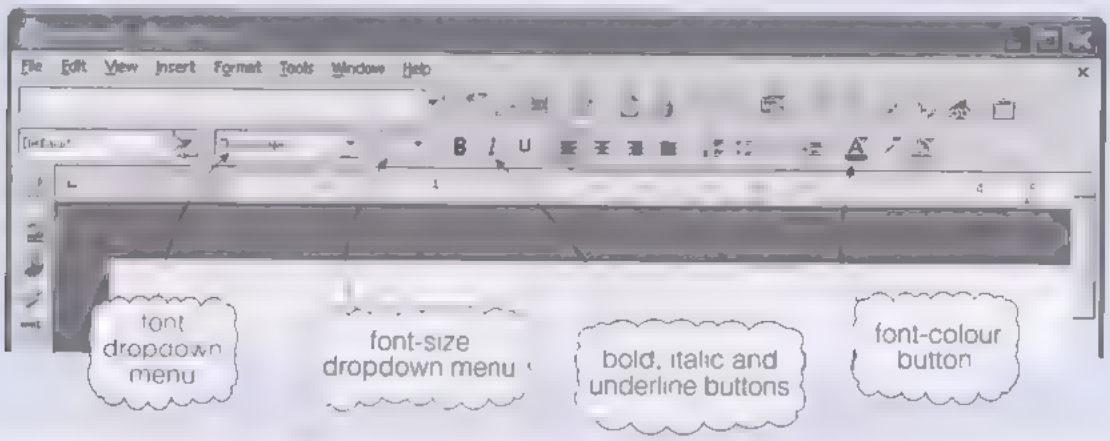


Figure 13 StarOffice text document toolbar

The font-colour button will change selected characters to the button's currently selected colour (see Figure 13) The button s selected colour can be changed by clicking on the button and holding the left mouse button down for a few seconds This will open up a colour palette as illustrated in Figure 14.

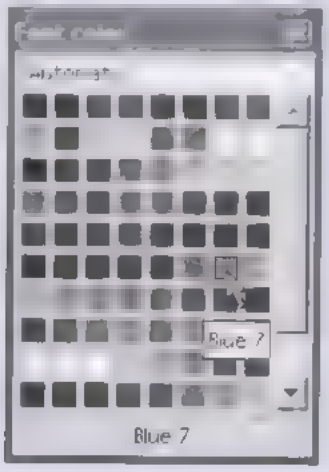


Figure 14 StarOffice colour palette

Click on one of the colours to change the colour button's selected colour, then close the palette and click on the colour button.

Type a few sentences into the StarOffice text pane. Then select pieces of text and do the following.

1 Change the style of one word (as a series of characters) to bold.
2 Change the style of another word to italic.
3 Change the colour of another word to green.
4 Increase the font size of some words and reduce the font size of others.
5 Change a word to a different font.

DISCUSSION OF ACTIVITY 4

When you typed text into the StarOffice text pane, StarOffice created character objects that have the attributes `colour`, `size`, `font` and `style`. When you selected a series of characters and then made them bold or italic you changed the values of each of those characters' attributes.

1 You set the `style` attribute of each of the selected character objects to `bold`.
2 You set the `style` attribute of each of the selected character objects to `italic`.
3 You set the `colour` attribute of each of the selected character objects to `green`.
4 You set the `size` attribute of each of the selected character objects to whichever point size you selected, for example `10`, `12`, `18`.
5 You set the `font` attribute of each of the selected character objects to a particular font, for example `Helvetica`.

In the next section we shall discuss the ideas that you met in the above activities, specifically that objects have attributes and that the values of these attributes constitute an object's state.

5.2 State

We shall now look at the practical work you carried out in Activities 1–4 in a more abstract way. In particular we shall consider which objects might have been used in such applications and what memory they needed so that they could provide the appropriate behaviour in response to the messages they received.

In the following discussion we shall describe everything in terms of a StarOffice drawing document. We shall not attempt to give a precise description of how it has been implemented, as we do not know. Rather we are using our knowledge of other object-oriented applications to give a plausible description. Most of the ideas can be applied equally to a StarOffice text document.

When you selected a shape such as a rectangle and changed its colour to, for example, red, the rectangle stayed red. It did not go back to its previous colour. This tells us that the object remembered the change. It has a colour attribute which was set to the new colour. Other changes to the rectangle, such as a change in position, persist in the same way. Each property of the rectangle is represented by an attribute and the values of the attributes record the current state of the rectangle. The same is true for other kinds of graphical object.

However, there is a bit more to it than just changing the values of the attributes. The shape objects are located on a drawing pane, and when a change occurs the drawing pane must be refreshed to show the new appearance of the object. What actually happens is something like this.

1 A rectangle's state is changed.
2 The drawing pane is sent a message saying the rectangle object has changed the value of one of its attributes
3 The drawing pane sends a message to the rectangle asking for details of the change
4 The drawing pane uses the information it gets back to redraw the rectangle showing the altered appearance

The idea behind all this is that it is the rectangle object itself that is responsible for knowing about its state. The drawing pane does not remember this information. If it needs the details it asks the rectangle. The same is true for all the other types of shapes.)

When an object is newly created what values do its attributes have? Some of its attributes may have *default* values. For example in the case of a rectangle in StarOffice the defaults are as follows

- fill colour – blue
- edge colour – black
- edge thickness – 0.0

All new rectangles share these values: their fill colour is always blue and they always have black edges of thickness 0.0. Other attributes of rectangle objects have values which are set by the user at the time the rectangle is created, for example

- position
- width
- height

The user chooses the position with the first click in the drawing pane, and the width and height by dragging. Typically different rectangles will be created in different positions with different dimensions. Of course, all this just applies to the initial values that the object starts off with. As you have seen, once an object already exists the values of its attributes can be changed.

When naming the attributes an object's class will have, you should give each a short descriptive name. Try to choose names which convey clearly what the attribute concerned represents. For example the meaning of the attribute called edgeThickness is instantly understandable.

Exercise 3

As you have already learnt in Java you have to run together multi-word identifiers, such as 'credit limit' into single words using a single upper-case letter to mark the start of each word after the first.

List all the attributes of rectangle objects, running together any multi-word identifiers and giving each a brief description.

Solution...

- `edgeColour` – colour of the edges of the rectangle
- `fillColour` – colour of the inside of the rectangle
- `edgeThickness` – thickness of the edge of the rectangle
- `width` – width of the rectangle
- `height` – height of the rectangle
- `position` – position of the rectangle

So you now have a list of attributes which you could use to describe a particular rectangle. You might say that a particular rectangle has the attribute ... set to the value `red`, the attribute `fillColour` set to the value `blue`, `edgeThickness` set to the value `4`, `width` set to the value `17`, `height` set to the value `9` and `position` set to the value given by column 22, row 25. Of course, the best way to represent such an object to humans who can perceive shape and colours, is to show it the way that StarOffice does ... for people with certain visual impairments speaking about the above description would be better.

So far when looking at text and word documents in StarOffice we have concentrated on state, which is made up of the values of an object's attributes. Now we shall start to consider the behaviour that ... when it is sent a message. In the next subsection we shall address these ideas in more detail.

5.3 Messages in a StarOffice text document

In this subsection you will look in some detail at what messages might be involved in changing the attributes of characters in a StarOffice text document. You will discover that a whole range of objects is involved, some of which are much less obvious than others.

As you have seen, it is possible to manipulate a sequence of characters – a word or paragraph ... exactly the same way, as you would a single character. You simply select what you want to change and apply the appropriate command ... Both flexibility and generality are reasons that word processors are now ubiquitous. We now look at how this aspect of word processors is made possible.

... StarOffice to alter the size of a selected piece of text, to say, 12 point. Or to change the characters to upright (roman) or italic? Consider the last sentence above. As a user you might select the text you want to change, then you click the Italic button. The effect is that the text you have selected changes to italic on the screen and, of course, if you were to print the document the changed text would appear in italics. Figure 15 depicts this.

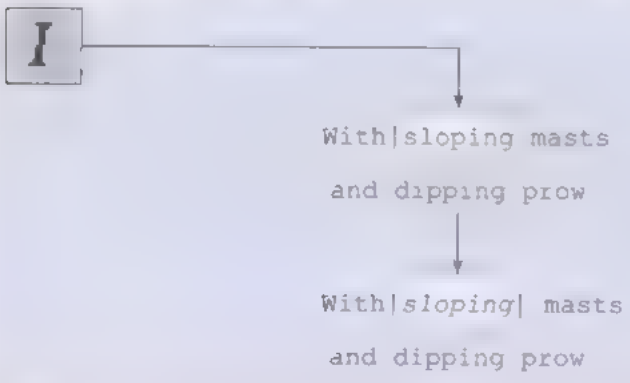


Figure 15 Changing characters to italic with the Italic button

The object-oriented view of computing considers the components of an executing program to be objects which know how to behave in some fashion and which enact their behaviour when sent appropriate messages. So if a text pane contained the text `...sloping masts and dipping prow`, what objects might be involved in producing `With` *`sloping`* `masts and dipping prow`?

The first object is the application itself. It needs to react to messages that the operating system sends it concerning mouse movement or clicks or typing on the keyboard. For example, is the cursor in the text pane, or is it somewhere on a tool bar (or somewhere)? Here the application needs to determine that the cursor is in the text pane (our second object) and tell the text pane to be ready for the user to do something. The text pane then needs to react when the user starts to select the characters in the word `sloping` by highlighting the characters. It must also remember which characters have been selected. The third object involved is the Italic button, which reacts to a 'mouse click' (the application is involved in this as well: the [illegible] tells it where the mouse is and that it has been clicked, and the application knows that the button object that it has been clicked). Fourth, and most important, there are the character objects whose **visual representation** [illegible] or other shapes `s l o p i n g`. These objects have a memory of their attributes – shape, font, colour, [illegible] and [illegible]. While these [illegible] visual representation, character objects in StarOffice almost certainly have other attributes whose values [illegible] important to themselves and to other objects in the application.

Once [illegible] has been selected and the Italic button has been clicked, a series of messages is sent. Firstly, the application is told by the button that the user has requested that some text be changed to Italic. The application then relays this information to the text pane. The text pane then checks if any characters are selected. If any are, the text pane then tells the selected objects to change to Italic. When the selected objects receive the message to change to Italic they do so. But that is not the end of the matter, the characters must tell the text pane that they have been [illegible] so that the text pane can redisplay them [illegible] to users that they have selected `sloping` and then Italic and the command has directly changed the word to *sloping*. All this is depicted in Figure 16.

While you should not [illegible] with the applications you use, [illegible] carry out with an application may involve lots of messages being passed among cooperating software objects to achieve what you want. You should be aware that the things you manipulate on screen are visual representations of software objects; they are not the objects themselves, which you cannot, of course, see.

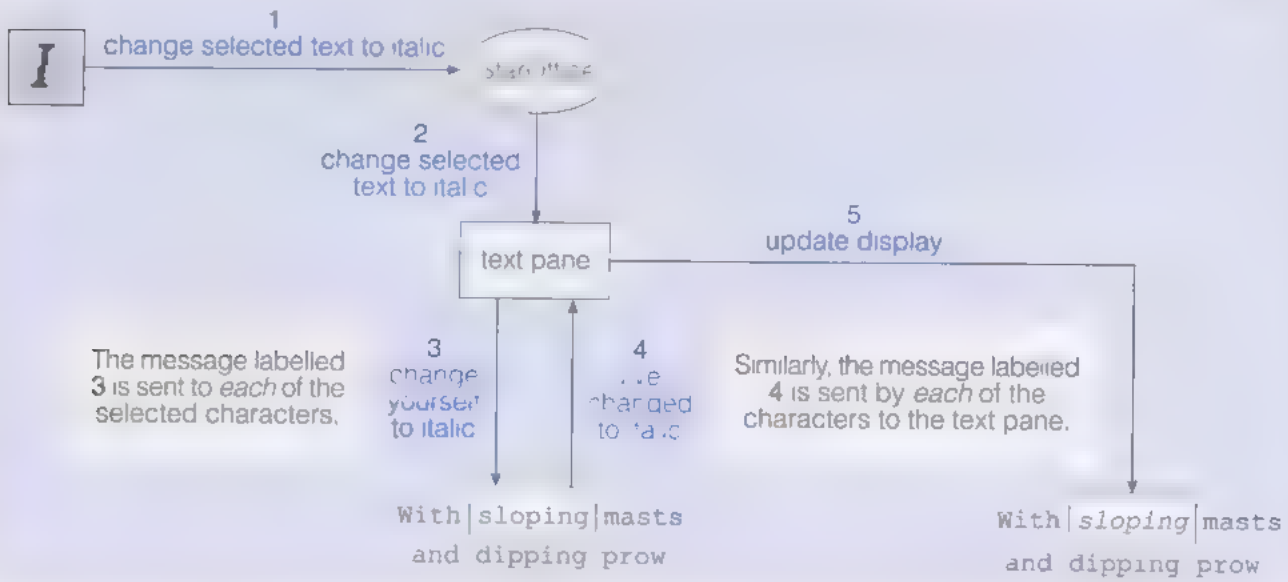


Figure 16 A model of how the Italic command works

6 Exploring objects in a microworld

Within the context of sending messages to objects, this section looks in more depth at the state of an object as given by the values of its attributes. The new concepts are those of an object being an instance of a class with a particular message protocol. We shall also introduce to you an application, developed specifically for the course, called Amphibian Worlds, that contains a series of microworlds containing amphibian objects. Most of the exploration of object-oriented ideas in this section will be carried out in the **microworld** Two Frogs. We will study other microworlds in part of *Unit 2*.

The activities start with the creation and sending messages to objects. From this exploration you will discover what attributes an object may have and how to use the software to inspect the state of an object at any given time.

You will also write your first piece of Java code so that you can manipulate objects via code rather than via a button in a user interface. You will then begin your journey into **object technology** by considering classes.

To access the microworld used in this section, launch the Amphibian Worlds application by double-clicking the shortcut that has been installed on your desktop and then, from the Microworld menu, select Two Frogs.

6.1 Sending messages to objects

A microworld is a computer-based simulation with opportunities for manipulation of content and practice of skills – in this case sending messages to objects and observing their effect.

The microworld Two Frogs has a user interface through which you can send messages to two frog objects either by clicking named message buttons or by entering Java code into a code pane and clicking the Execute button. The frog objects also have graphical representations so you can observe the effects of sending messages to them. The Inspect button in the microworld will enable you to look inside an object to find out its state – the values of its attributes.

Figure 17 The microworld Two Frogs

The graphics pane at the top of the microworld Two Frogs displays graphical representations of two frog objects. These frog objects are shown sitting on stones in a pond.

Variables are discussed in detail in *Unit 3*.

In order to send messages to these objects you need some way of referring to them. For this we use variables. A **variable** is a named block of computer memory into which data can be stored. In the case of the [illegible] Frogs there are [illegible] variables [illegible] and [illegible]. These variables are [illegible] below the graphics pane. These [illegible] pleased, for example x and y, or `aObject` and `bObject`. However, using `frog1` and `frog2` is a better choice because the names remind you what sorts of objects are involved. The buttons to the right of the list of variables in the pane labeled [illegible] Messages) allow you to send messages to the two frog objects and observe their behaviour in the graphics pane. However before sending a message it is necessary to indicate first to the microworld which frog object you wish to send the message to. To select a particular frog object you will need to highlight the appropriate variable in the list pane by clicking once on it. Once you have selected a variable the graphics pane will identify the graphical representation of the corresponding frog object by colouring its stone yellow. Having selected a frog object, you will then be able to send it a message by selecting (clicking once on) a message button.

At this point it is useful to clarify the terminology associated with variables. You may come across informal shorthand phrases such as 'send a message to the object `frog1`' or 'as before `frog1` behaved as expected'. However since a variable is just a label on a block of memory, to be technically precise the phrases should be 'send a message to the object *referenced by the variable* `frog1`' and 'as before *the object referenced by the variable* `frog1` behaved as expected'. Most programmers would be perfectly happy using the shorthand style in casual speech (and indeed we will occasionally use it in this and subsequent units) but you should always be prepared to use the more precise terminology when required.

ACTIVITY 5

Launch the application Amphibian Worlds and then select the microworld Two Frogs from the Microworld menu.

We want you to explore the behaviour of the two frog objects in this microworld a few suggestions and questions for working systematically are listed below You may find it useful to record the behaviours of the frog objects in response to each message To send the message to the object referenced by highlight the variable in the Amphibians list pane and then click once on the button labelled right() The representation of the object in the graphics pane will then move one position to the right (to a new stone).

1 What are the colour and position of each frog object when you first open the microworld Two Frogs?

2 Select one of the frog objects (highlight a variable in the list pane) and send it each message in turn (by clicking once on a message button), noting the response An error report will appear in the window labelled Display Pane when you send some of the messages Read the text and then press the Clear button to remove the error report List the messages the frog object understands and record the response each such message invokes in the frog object.

3 Does the other frog object respond in the same way to each message?

4 With a frog object sitting on the rightmost black stone, send it the message a few times in succession you will see a blue arrow What is the blue arrow indicating? Try to reposition the frog object on the rightmost stone by sending messages to the frog object.

With a frog object sitting on the leftmost black stone send it the message a few times you will see a red arrow What is the red arrow indicating? Try to reposition the frog object on the leftmost stone by sending it messages

5 What happens when the message is sent to a frog object? What happens when the message down() is sent to a frog object?

Following from your explorations of the messages to which the frog objects respond and their resultant behaviour what information do you think each frog object is storing? Can you make any guesses about the attributes of these frog objects and the state a particular frog object may have?

DISCUSSION OF ACTIVITY 5

1 When the microworld Two Frogs is opened each frog object is green and is on the leftmost stone (position 1)

2 Whichever frog object you select it responds to the following messages with the following behaviours

`left()` – moves one position to the left

`right()` – moves one position to the right,

`home()` – moves to (or remains on) the leftmost black stone,

`jump()` – jumps, and lands again in the same position,

`green()` – turns green (unless already green),

`brown()` – turns brown (unless already brown),

`croak()` – croaks audibly (and displays a red !)

3 The two frog objects behave in exactly the same way when the same message is sent to them. For example both objects move one position to the right when sent the message `right()`.

4 If a frog object is on the rightmost black stone is sent the message `right()`, the graphics pane shows a blue arrow pointing to the right to indicate that the frog object has disappeared from view. If a frog object on the leftmost black stone is sent the message `left()` the graphics pane shows a red arrow pointing to the left to indicate that the frog object has disappeared from view.
If either a blue or a red arrow appears, then a horizontal scrollbar will appear underneath the graphics pane allowing you to scroll the graphics pane left or right to view the frog object that has gone out of sight. Alternatively a message to the frog object to move in the opposite direction will cause it to reappear. The frog object can also be repositioned to the leftmost black stone (position 1) by selecting the variable that references the frog object in the list pane and pressing the home() button.

5 When the message up() or down() is sent to a frog object, a pane – the Display Pane – opens up at the bottom of the Methods window and a message in the pane appears to inform you that an error has occurred. This is because you are *not allowed to* send up() and down() to ordinary frog objects, which are not capable of acting on these messages. These messages can be sent to a more versatile kind of frog that you will investigate later, in *Unit 2*! To close the Display Pane click on the – sign next to the words Display Pane.

The messages [illegible] affect the position of a frog object. (You see the icon representing the frog object move to – or remain on – a particular stone.) The object must therefore hold information on its position so it is likely to have an attribute with a name like [illegible]. In fact the attribute [illegible] holds a number reflected in the graphics pane by mentally numbering the stones from left to right with the frog object appearing on the leftmost black stone when the attribute's value is 1, and the frog object appearing on the rightmost black stone when the attribute's value is [illegible]. As you have seen in the previous activity, frog objects can move to positions outside the range of 1 to 11. When this happens a red or blue arrow appears in the graphics pane to indicate that the frog has moved out of sight, but you can scroll the graphics pane to bring the [illegible]. Stones representing positions less than one are coloured light red, and stones representing positions greater than 11 are coloured light blue.

A frog object's colour can be [illegible]. (The icon changes colour [illegible].) This reflects the object's state and you might guess that there is an attribute with a name like `colour`. This attribute must be able to specify the colour of the frog object, including the colours green and brown.

ACTIVITY 6

In the previous activity you made a guess at the attributes (and their values) of the frog objects. Now you are going to 'look inside' each object to see what attributes each has been given by the programmer. The formal term for 'look inside' is to *inspect the state of an object*. The microworld provides an inspector tool for finding out an object's attributes and for inspecting an object's state (the current values of the attributes). Figure 18 shows an example of the inspector tool displaying the attributes and attribute values of a frog object.

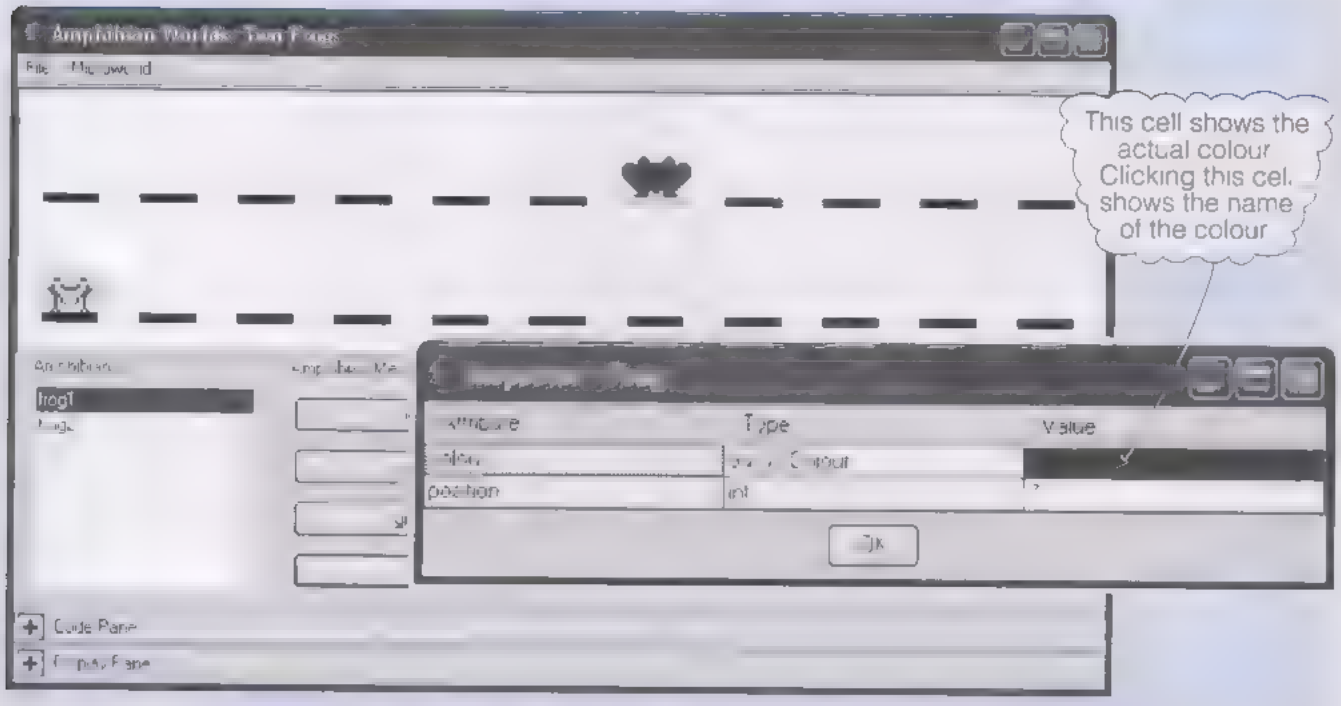


Figure 18 An inspector on a frog object

1 Highlight the variable frog1 in the list of variables and press (click once on) the button labelled Inspect. An inspector will open on the object referenced by the variable frog1, enabling you to look inside the object that is to ascertain the attributes of the object referenced by `frog1` and its current state.

What attributes has the object referenced by frog1? How would you describe the current state of the object referenced by `frog1`?

Close the Inspector window before proceeding.

2 Now inspect the attributes of the object referenced by frog2, following the approach given above for frog1. Do the two frog objects have the same attributes? Do the frog objects currently have the same state?

How can the state of a frog object (or, in general, of any object) be changed?

Close the Inspector window before proceeding.

DISCUSSION OF ACTIVITY 6

The left hand column of the Inspector window gives the attributes of the object being inspected and the right-hand column gives the attribute values – its state.

1 You can see that `frog1` has two attributes – `position` and `colour`. If the frog object is in its default position (the leftmost black stone) and its original colour (green) then the inspector will show the state of `frog1` to be `position` 1 and `colour` `GREEN`. Note that when you click on the value of the `colour` attribute the text name for the colour will be shown rather than the colour itself.

2 Both frog objects have the same attributes. They may or may not have the same attribute values, this will depend on what messages have been sent to them. If `frog1` has `position` set to 1 and `colour` set to `GREEN`, and `frog2` has `position` set to 2 and `colour` set to `BROWN`, the states of the two frog objects are not the same.

The state of an object can be changed by sending it a message.

An inspector shows a snapshot of the state of an object not a live report If you leave an inspector open on say and send a message to that changes its state and then return to the open inspector window this inspector does not reflect the new state In order to see the new state of it is necessary to open another inspector The new state of the object referenced by is then displayed in the new inspector

ACTIVITY 7

The microworld Two Frogs includes a Code Pane Instead of clicking the various named message buttons to send messages to frog objects you can use the Code Pane to write Java code that will send messages to these frog objects once you click on the Execute button.

To try this out, firstly click on the + sign next to the words Code Pane (just above Display Pane) to open the pane up, then place the cursor in the Code Pane type the name of a variable that references a frog object and then the message you wish to send *Take care to place a full stop between the variable and the name of the message because, in the syntax of Java a full stop is how you indicate that a message is to be sent* Make sure that you copy the capitalisation of each letter exactly as upper or lower case, and complete your message with a semi-colon An example line of code is

```
frog1.brown();
```

If you make a typing mistake you can correct it much as in a word processor Once you are happy with what you have typed press the Execute button The effect of your message will be shown in the Graphic Pane.

If you send a message with a typing mistake in it an error report will appear in the Display Pane If the advice does not help try typing the message again making sure you have deleted everything you don t want before retyping Pay particular attention to spaces (you do not need any) capitalisation (use of upper and lower case) and spelling as these are the most common typing errors. Highlight and delete your message line each time before typing the next

Try typing a variety of messages to each frog object including the messages and `down()`.

DISCUSSION OF ACTIVITY 7

Sending messages to objects using the code pane produces exactly the same results as sending the same messages by selecting a variable in the list pane and clicking a button

In your work in the microworld Two Frogs you learnt that you need a way of referring to an object a variable before it can be sent a message You then sent messages to the two frog objects and observed the results of those messages the behaviour of the frogs in response to the messages. You did this in two ways

- Firstly by selecting a variable in the list pane and then clicking an appropriate message button to send a message to the frog object referenced by that variable
- Secondly by typing the Java code into the Code Pane and then clicking the Execute button When you did this you had to ensure that you spelled the name of the variable and the message correctly and used the correct syntax Using the Code Pane acted as your first exposure to the Java programming language.

In both cases you observed the results of sending the messages in the graphics pane of the microworld which displayed graphical representations of the frog objects You also

checked the state of the frog objects by first selecting an appropriate variable from the list pane and then clicking the Inspect button to open an inspector which displayed the attributes and attribute values of the object referenced by that variable.

We call the object that is sent a message the **receiver** of the message, as shown in Figure 19

Figure 19 The message `left()` being sent to the receiver `frog1`

The [illegible] sending a message (as shown in Figure 19) is called a **message-send**

SAQ 3

In the following code, indicate the message and the object receiving the message

```
frog2.brown()
```

ANSWER

The message `brown()` is sent to the object referenced by the variable `frog2` – this object is the receiver

6.2 Grouping objects into classes

You have discovered that the variables [illegible] and [illegible] refer to very similar objects. In fact, these objects respond to exactly the same set of messages, have the same attributes, and behave in exactly the same way in response to the same message.

As well as saying that the 'object referenced by x belongs to class A', we also use the synonymous phrase 'x refers to an instance of class A'.

These similarities occur because the objects referenced by [illegible] and [illegible] belong to the same **class**. When using an inspector to look inside a frog object in the microworld Two Frogs, the Inspector window has the title Inspector: Frog.

The two frog objects in the microworld have been created as **instances** of the [illegible] class. A class is like a blueprint or template for the creation of objects and ensures that all its instances have the same attributes, and respond to the same set of messages in an identical manner. So two different objects that belong to the same class and are referenced by the variables `frog1` and `frog2`

- understand the same messages;
- respond in the same way to each message;
- have the same attributes.

On creation, the objects referenced by the variables [illegible] and [illegible] each has its own set of attributes and their states are the same – they both have the attribute [illegible] with value 1, and the attribute `colour` with value `GREEN`. Each object has its own independent *copy* of these attributes, so that it can remember its own individual state because although all new [illegible] objects are created with identical state, the state of any particular frog will get altered during its lifetime when it is sent messages such as

`right()` or `brown()`. For example, at a later time, `frog1` may still have a value of 1 for its attribute `position`, whereas `frog2` may have this attribute set to 3.

SAQ 4

'Instances of a given class have the same attributes.' Explain this statement.

ANSWER

A class defines what attributes each instance of the class will have. For example, `frog1` and `frog2` are instances of class `Frog` and have the attributes `colour` and `position`. However, each instance of the class responds to the message `position()`, is sent to `frog1` and the message `position()` is sent to `frog2`, the value of each object's attribute `colour` is different.

6.3 Grouping messages into a protocol

The set of messages to which any instance of the class can respond is called its **protocol**. (Strictly speaking, the set of messages to which instances of a class can respond is known as the **instance protocol** of the class.)

The protocol of a `Frog` object, as we know it so far, is `left()`, `right()`, `home()`, `jump()`, `green()`, `brown()` and `croak()`.

SAQ 5

Use the terms class and protocol to explain why a `Frog` object is able to respond to the messages `up()` and `down()`.

ANSWER

The class defines the set of messages (the protocol) to which an instance of that class can respond. `Frog` objects are instances of class `Frog` and the instance protocol for this class does not include the messages `up()` and `down()`.

6.4 Attributes of frog objects

The only attributes a `Frog` object has (in the microworld we have been exploring) are `colour` and `position`. The values (or examples) to which these attributes are set are shown in Figure 20 and these values constitute the state of a `Frog` object.

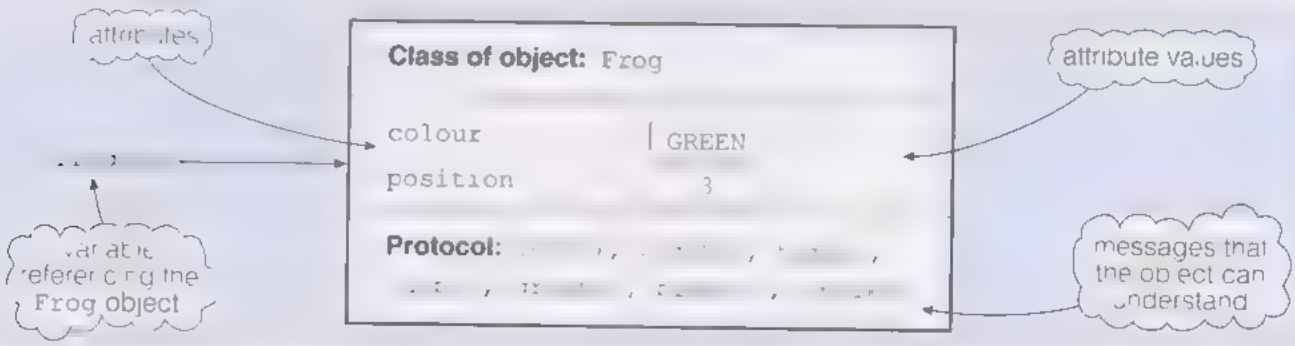


Figure 20 Diagrammatic representation of a `Frog` object

The only way that the state of a Frog object can be changed is by sending it a message. The value of the attribute colour can be set to GREEN or BROWN. The value of the attribute position can only be a number. (This number is reflected in the graphics pane of the microworld. Two Frogs are on the stone on which the object corresponding with 0 being the leftmost black stone and 11 the rightmost black stone.)

SAQ 6

What is the state of the Frog object depicted in Figure 20?

ANSWER

The state of the Frog object is position set to 3 and colour set to GREEN. More colloquially you might say something like 'It's in position 3 and its colour is green'.

SAQ 7

[illegible] the object referenced by [illegible] has been sent the message [illegible] what is its state after responding to the message?

ANSWER

The state of the Frog object referenced by [illegible] After it has responded to the message [illegible] and the value of its attribute colour is BROWN.

6.5 Messages that do not alter an object's state

Most of the messages [illegible] object's state – either their colour or position. However, it is quite common to come across messages that do not cause any state change. As an example, we [illegible] the message [illegible] in the project of [illegible] objects. Sending the message [illegible] Frog object makes it send another message to the microworld to tell it to [illegible] graphically jumping but leaves its state unaltered. You can tell that the state is the same, because the only attributes Frog objects have are colour and position, and neither is affected by the message jump().

Later in Unit 2 we discuss messages that query an object's state in order to return the value of a particular attribute. Such messages do not usually change the state of the receiver.

7 Classes as software components

Now that you have had a chance to discover some of the characteristics of objects, it is a good time to consider why object technology has become so important to the software industry.

When forming a new product in traditional industries, such as car manufacturing, the designer no longer designs a unique, hand-crafted artefact, down to individual nuts and bolts. The designer can take advantage of ready-made sub-assemblies (components), for example a gearbox from Honda, an engine from Mazda, a fuel injection system from Bosch, suspension from Lotus and a body shell from Pininfarina. All these companies will have ensured that the fixing brackets for their products are of a standard size and that they have holes pre-drilled to accept standard sized nuts and bolts.

Until the early 1990s the software industry was more like the early car manufacturing industry with each part of an application or system being designed from scratch. More recently, standard **software components** have been produced in a similar fashion to car components with the aim of reuse. The same reliable, reusable software components can be incorporated into many whole new systems, thus saving the considerable time and effort it can take to generate new software. These possibilities have been brought about by object technology.

A growing part of the software industry is now focused on the production of generally useful software components (and at the other end of the scale, highly specialised software components) that can then be bought by other software developers to speed the development of their own applications (thus avoiding 'reinventing the wheel'). For example, if a company were to write a system for an online shop, it is very unlikely that they write all the code from scratch. It is more likely they would buy a database component from one vendor (to hold details of [illegible] items for sale) and a component for secure payment transactions from another vendor.

The reason why this is now possible is because objects are entities that contain both data (in the form of attribute values) and a defined message protocol. To repeat a mantra – objects only do something if you send them a message. They are self-contained units of software that can be tested and proved to be robust and reliable. In the context of object-oriented programming, a software component is a class, or more likely, a closely related group of classes. (Note that the **component** is the class (not the object): what you buy, as a component, is the code for constructing and using instances of a class, not the instances themselves.)

The concept of software components has led to the possibility of replaceable parts for systems – not just for replacing faulty components with correct versions, but for replacing limited components with more flexible ones. For example, imagine a component of a word processor, say, one that allows the word processor to manipulate documents. If the relevant component were only to accept documents of less than 6000 characters (merely a couple of pages), but you wanted the word processor to be useful for writing a book, it would be helpful to replace the original limited component with one that accepted, say, 2 400 000 characters. This is entirely possible so long as the classes in the new component defined objects with the same protocol. Just as your garage can simply replace the engine in your car with a more powerful one if the fixing brackets are in the same place.

SAQ 8

Why are the components of a domestic electrical system (such as plugs and light bulbs, a suitable analogy for the ideal software industry?

ANSWER

It is a suitable analogy because the domestic electrical system depends on standard parts – you can exchange different makes of each component such as a plug or a bulb (at least within one country) and the system will still work. As long as each component works as intended, its make is irrelevant.

8 Summary

After studying this unit you should understand the following ideas

- Ultimately software executes on hardware: software delivers instructions to hardware
- Types of software can be categorised as system, application or program software (however these categories do overlap to some extent)
- An operating system (OS) is the software responsible for the control and management of hardware, and the basic computer system operations.
- Source code must be translated by a compiler into a primitive language called machine code, in order to run on your computer's hardware.
- The translation of source code to machine code can be a two stage process. Firstly it is compiled to bytecode which is the machine code of a virtual machine. This virtual machine will then interpret the bytecode into machine code at run-time
- The advantage of a compilation mode that makes use of a virtual machine is that it ensures that the bytecode, no matter on what machine it was compiled, can be translated for execution on many different computers, so long as each computer system has the correct virtual machine installed
- In object oriented software all the processing that is carried out by a program is done by 'objects'
- An object can be thought of as a self-contained unit of software that holds data and knows how to process that data
- The only way to get an object do anything is to send it a message
- All messages ask an object to perform some action. These actions constitute what is termed the behaviour of the object
- The set of messages to which an object responds is called its protocol
- An object has attributes. The values of these attributes at any one time constitute the state of the object
- A message may change the state of an object.
- A message may make an object do something without altering its state.
- Objects are organised into classes. A class defines the attributes and behaviour of its instances. Therefore objects belonging to the same class (instances of the class) have the same set of attributes and respond to the same set of messages, responding to each message in an identical manner
- Object technology has brought about the concept of software components which are produced with the aim of reuse. The same reliable, reusable software components can be incorporated into many whole new systems, thus saving considerable time and effort in producing new software

LEARNING OUTCOMES

After studying this unit you should be able to:

- explain the differences between hardware and software;
- categorise examples of software as systems, applications or programs;
- describe the role of the operating system;
- explain various methods for translating source code into machine code;
- describe the role of Java Virtual Machine;
- describe how the advent of the World Wide Web contributed to the success of Java;
- appreciate and describe what characterises object-oriented software;
- explain how procedural software differs from object-oriented software;
- explain the terms object, attribute, state, operation, message and object as they apply to objects;
- make sensible choices about what objects might be used in a piece of software and the sorts of message to use with those objects;
- give an account of what happens when a user uses a button or a menu to change the appearance of something on screen;
- reason about what attributes a particular object might have and what values those attributes might have at a given time;
- describe how objects are organised into classes which determine what attributes an object has and to which messages they can respond;
- explain how object technology supports the possibility of building software systems out of components.

Glossary

A mini-glossary highlights the key terms in the unit. Some of these terms are developed further in subsequent units and also are presented in more detail in those units' glossaries.

application Software that turns your computer into a specialised computer such as a word processor or web browser.

application domain See **problem domain**.

attribute Some property or characteristic of an object, such as `position`, `size` or `balance`.

attribute value The current value of an attribute.

behaviour Used to describe the [illegible] of an object.

binary digit Either of the two digits 0 or 1. Binary digits are used for the internal representation of numbers, characters and instructions. The binary digit is the smallest unit of storage.

bit See **binary digit**.

bytecode The code produced by a compiler as the machine code of a virtual computer. Bytecode is so-called because it is organised into 8-bit bytes.

class A class is a template that serves to describe all instances (objects) of that class. It defines what **attributes** the objects should have and their **protocol** – what messages they can respond to.

compiler A program that translates a program into **bytecode** or machine code.

component See **software component**.

domain See **problem domain**.

domain model The part of the software that models the **problem domain** as opposed to being directly concerned with how communication with the user is achieved.

dynamic compilation A compilation technique used by the Java environment, generating real machine code from **bytecode** (intermediate code). A chunk of bytecode is compiled into machine code just prior to being executed. This is different from and faster than the perennial operation of an interpreter. The real machine code is retained so that subsequent execution of that chunk of bytecode does not require the translation to be repeated.

high-level language A language (for example Java) whose structure reflects the requirements of the problem rather than the facilities actually provided by the hardware. It enables a software solution to a problem or a simulation of an aspect of reality to be expressed in a hardware-independent manner.

instance An **object** that belongs to a given **class** is described as an instance of that class.

intermediate code See **bytecode**.

just-in-time compilation See **dynamic compilation**.

low-level language A language written for direct programming of a computer's hardware. Each type of computer hardware needs its own low-level language

message A request for an object to do something – `right()` is an example of a message. The only way to make an object do something is to send it a message

message-send The code that sends a message to an object – for example, `frog1.right()`, which consists of the **receiver** followed by a full stop and then the **message**

microworld A computer-based simulation with activities for manipulation of content and practice of skills

model (verb) To simulate an entity in the **problem domain**

model (noun) See **domain model**

network computing [illegible] such a 'network'. A network may be local (for example, in an office or home) or global (for example, the Internet)

object A **software component** [illegible]

object-oriented technology [illegible] being made up of objects

object technology A synonym for object-oriented technology

operating system The **software** [illegible] maintaining security, and detecting equipment failure

peripheral device Any part of the computer that is not part of the essential core (i.e. the CPU and main memory). The most common peripherals are input and output (I/O) devices such as the mouse and keyboard, and storage devices such as hard drives and CD/DVD drives

problem domain The collection of real-world entities within the application area that exhibit the behaviours that the required system has to model

program **Software** that has a start, [illegible] that takes some input, after which it performs whatever computation is needed, and has an end point at which output is given and the software ceases to run

protocol The set of **messages** an **object** can respond to (understands)

receiver The **object** to which a **message** is sent

run-time Refers to the moment when a **program** begins to execute. In contrast to the time at which it has been loaded or compiled

run-time system The code that a **compiler** produces to make software execute on a real or **virtual machine**. This code has not been explicitly written into the source code by the programmer

software A general term for all the **applications**, **programs** and **systems** that run on your computer

software component A piece of software that can be combined with other pieces to construct software

source code Program text expressed in a high-level programming language.

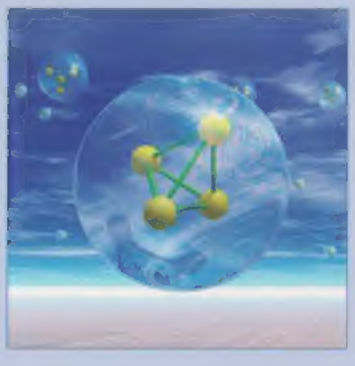
state The state of an **object** is the information it needs to implement the behaviour that its protocol requires. The object's state is determined by the values of its attributes.

system Software that is intended to run forever, responding to events in often complex ways.

virtual machine A layer of software that simulates a computer capable of interpreting **bytecode**.

visual representation A useful representation of a software object which, by definition, is invisible. The visual representation may be textual, such as characters in a word processor; or graphical, such as shapes in a drawing application.

Index



M255 Unit 1
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

UNIT 1

Block 1

M255 Unit 1
ISBN 978 0 7492 5493 3